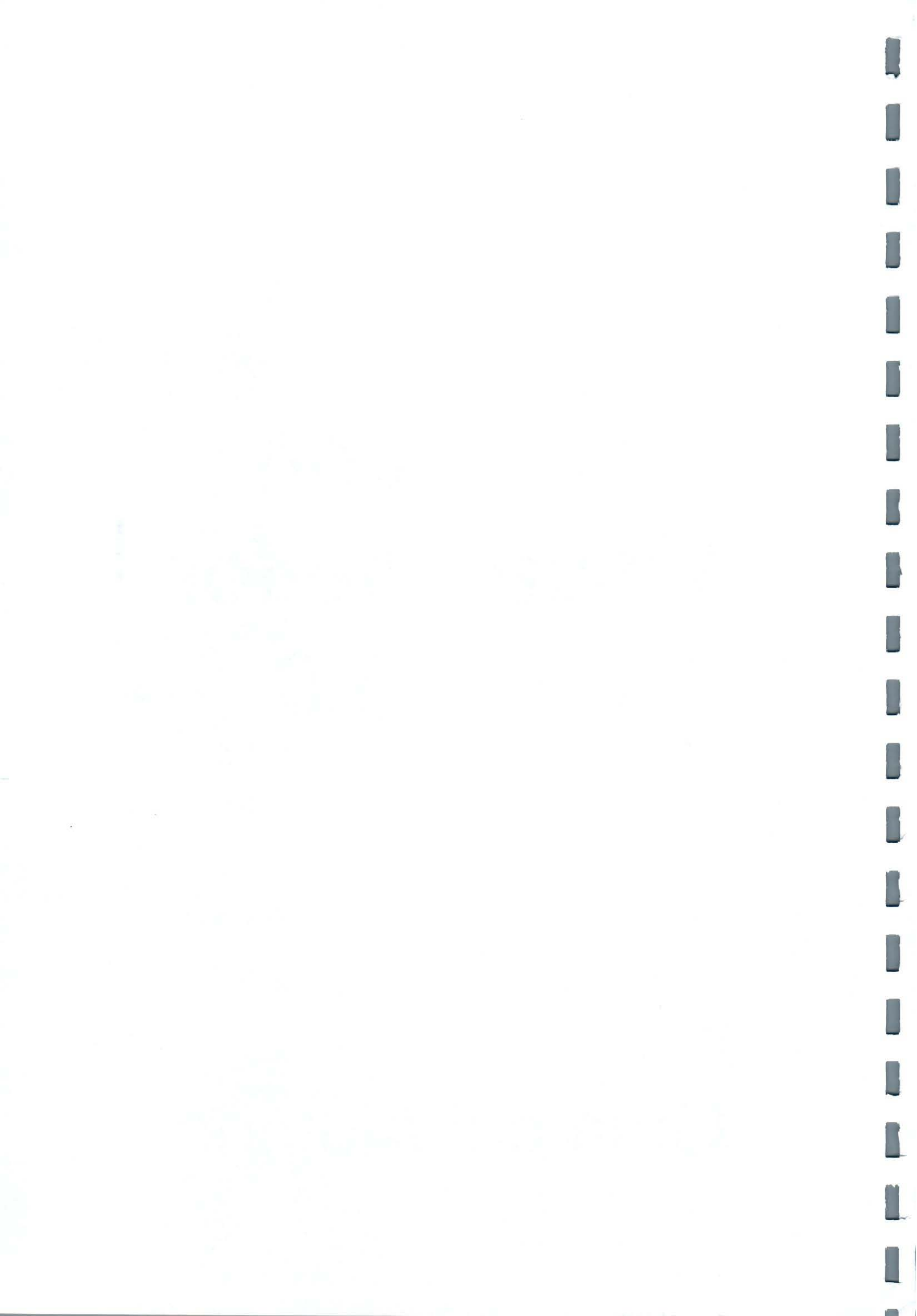
OOP con VisualAge for Java 3.0

Guía del Alumno



Contenido de los Capítulos

1 Introducción, Características de Java, Visual Age for Java

2 Tecnología y Conceptos de Orientación a Objetos

Facilidades para protección de acceso y otros Modificadores
Visibilidad de miembros desde el exterior de la clase
Modificadores de acceso para miembros
public : Modificador de acceso para clases
static : Variables de clase
static : Métodos de clase
final : La versión final de variables y métodos
final : La versión final de una clase
abstract : Implementación de clases abstractas
synchronized, native, transient, volatile
Modificadores : resumen y símbolos usados por Visual Age for Java

5 Interfaces y Clases internas
Bibliotecas de clases
El paquete java.lang

Necesidad de clases en la compilación
Necesidad de roles y "herencia múltiple"
interface : Declaración de Interfaces
implements : Implementación de Interfaces
Clase Internas
Paquetes y directorios. Unidades de compilación
Biblioteca de clases
java.lang : el paquete núcleo
La clase Object
Las clases para "envoltura" de tipos primitivos
La clase String
La clase StringBuffer

6 Clases para Interfaz gráfica al Usuario (GUI) :
El paquete java.awt
El paquete java.swing
Programación con eventos :
El paquete java.awt.events

java.awt (Abstract Windowing Toolkit): componentes para GUI
Componentes simples
Componentes de texto (TextComponent)
Contenedores (Container)
Window

3 El lenguaje Java
Tipos de datos simples y estructuras de control

El lenguaje Java
Identificadores
Identificadores reservados
Literales
Operadores
Separadores y Comentarios
Declaración de variables con tipos de datos primitivos
Arreglos
Ámbitos o bloques (estructuras secuenciales)
if : Ejecución condicional
if - else : Ejecución condicional dicotómica
switch : Ejecución condicional múltiple
for : Iteración (bucle)
while : Iteración (bucle)
do - while : Iteración (bucle)
break : Bifurcación incondicional
continue : Bifurcación incondicional

4 El lenguaje Java
Tipos de datos compuestos
Modificadores para clases y miembros

Clases y Objetos
Variables de Instancia
Métodos. Declaración de métodos
Invocación de métodos
Declaración de clases (Tipos compuestos) y variables de referencia
El método constructor y la creación de instancias (objetos)
El operador . (punto)
Herencia : derivación de subclases y variables especializadas
Herencia : métodos especializados
Polimorfismo : Sobreescritura de métodos (Overriding)
Selección de métodos en forma dinámica
Polimorfismo : Sobrecarga de métodos (Overloading)
this y super : Variables de referencia especiales
Paquetes.
Importación de paquetes

Menús (MenuCompoent)
Programación con eventos
El modelo por herencia para manejar eventos
El modelo por delegación para manejar eventos
java.awt.events : Clases de eventos
Listeners y Adapters (Interfaces y facilitadoras para manejo de eventos)
Clases para ubicación y ajuste de componentes
Una aplicación con eventos y componentes
java.swing

7 Applets y Aplicaciones

La clase Applet
Applets y Aplicaciones
Los métodos de un applet
Ejecución de un applet
La especificación en HTML
Facilidades de Visual Age for Java para desarrollar applets
Implementación de un applet

8 Manejo de Errores y Excepciones

Manejo de Excepciones
La clase Trowable
La clase Error
La clase Exception
Excepciones en operaciones de Entrada/Salida
Un programa con error
try y catch : capturando y manejando excepciones
try anidadas
throw : lanzando excepciones
throws : lanzando excepciones no capturadas
finally : independizando código de las capturas

9 Hilos y Sincronización (Multithreading)

Programación con múltiples hilos
El cuerpo de un hilo
Los estados de un hilo
Programa de un sólo hilo
Programa de varios hilos
La interfaz Runnable
La clase Thread

Integridad Referencial
Restricciones y Acciones de Integridad Referencial
SQL
SQL : Lenguaje de Definición de Datos (DDL)
Tipos de datos SQL
Diseño Físico de Bases de Datos
SQL : Lenguaje de Manipulación de Datos (DML)
SELECT : Recuperación (consulta) de filas
INSERT : Inserción de una o más filas
DELETE : Eliminación de una o más filas
UPDATE : Actualización de una o más filas
SQL : Lenguaje de Control (DCL)
Control de Acceso a los datos
Niveles de Seguridad
Privilegios sobre objetos de base de datos
Control de Transacciones sobre bases de datos
Necesidad de bloqueo de recursos
Niveles de Aislamiento (Isolation Levels)

13 JDBC

java.sql : el paquete JDBC
La clase DriverManager
La clase Types
JDBC categoría 1 : El puente JDBC-ODBC
JDBC categoría 2 : JDBC- driverDelDBMS
JDBC categoría 3 : JDBC - driver de red
JDBC categoría 4 : JDBC nativo (el ideal)
Estructura de una aplicación JDBC
Carga del driver
Conexión con la base de datos
Ejecutar sentencia SQL
Manejar el resultado de una sentencia
PreparedStatement : Sentencia para SQL dinámico

14 Data Access Builder

Visual Age for Java versión Enterprise
JDBC y formas de utilizarlo
El Data Access Builder
Desarrollo de aplicaciones con Data Access Builder
La tabla DEPARTAMENTO

10 Beans

Componentes de Software
Beans de Java
Características y Tipos de Beans
Integración con herramientas visuales
java.beans : el paquete del API de JavaBeans
manejo de propiedades de un bean
Introspección
La clase Class
java.lang.reflect :el paquete para introspección
Construcción de un bean

11 Programación Visual

Introducción
Construcción de aplicaciones y applets con beans y eventos
El Editor de Composición Visual : la herramienta de VAJava
Arrastrando beans : definiendo la GUI de la aplicación
Incorporando beans no visuales
Conexiones Evento-a-Método
Conexiones Parámetro-de-Propiedad
Conexiones Fuente-Vista (de datos)
Conexiones Evento-Código

12 Bases de Datos

Bases de Datos
Sistemas de Administración de Bases de Datos
Plataforma Cliente/Servidor para uso de DBMS's
El modelo Relacional para Bases de Datos
Estructura del modelo Relacional
Operaciones del modelo Relacional
Diseño Lógico de Bases de Datos
Modelamiento de datos : Relaciones y cardinalidades

Contenido de los Laboratorios

Laboratorio 1.- El Ambiente de Desarrollo Integrado del Visual Age for Java

- El ambiente de desarrollo
  - El IDE
  - (~~Integrated~~ Intelligent Development Environment de Visual Age for Java)
- Elementos de programas
  - Proyectos
  - Paquetes
  - Clases
    - Applets
  - Métodos
- Edición de elementos de programas
- La ventana "Consola" (Console)
- La ventana para pruebas pequeñas (Scrapbook)
- Búsqueda y aprovechamiento de las clases provistas
  - La clase Math

Laboratorio 2 .- Tipos simples. Estructuras control

- Programas ejecutables
  - método main
  - Argumentos de un programa ejecutable
- Operadores
  - operadores enteros
- Estructuras de control
  - if
  - while
  - do - while

Laboratorio 3.- Tipos compuestos : Clases. Herencia

- Programas ejecutables
  - método main
- Clases
  - Método constructor
  - Sobrecarga de métodos
  - this

OutputStream
FileDescriptor
FileInputStream
FileOutputStream
BufferedInputStream
BufferedOutputStream

Laboratorio 8.- Hilos y Sincronización (Multithreading)

Hilos
- clase Thread
- interfaz Runnable

Sincronización
- modifcador syncronized

Comunicación entre hilos
- método wait()
- método notify()

Laboratorio 9.- Beans y Programación Visual

Beans
El Editor de Composición Visual
Programación Visual
- Conexiones

Laboratorio 10.- Bases de Datos y JDBC

Bases de datos
- Creación de una base de datos
- Creación de una tabla
- Carga de datos con Exportación

JDBC
- Carga de driver JDBC
- Conexión a una Base de datos
- Ejecución de sentencias SQL
- Barrido de resultados extraídos de bases de datos

Herencia
Invocación de métodos

Laboratorio 4.- Tratamiento de cadenas

Sentencias condicionales
- if
- switch

Sentencias para Bucles (loops)
- for

Tratamiento de cadenas de caracteres.

Laboratorio 5.- Programación con eventos : El paquete java.awt.events

Programación con Eventos
- Eventos del mouse

Applets
- clase Container

Gráficos
- clase Graphics
- clase Rectangle

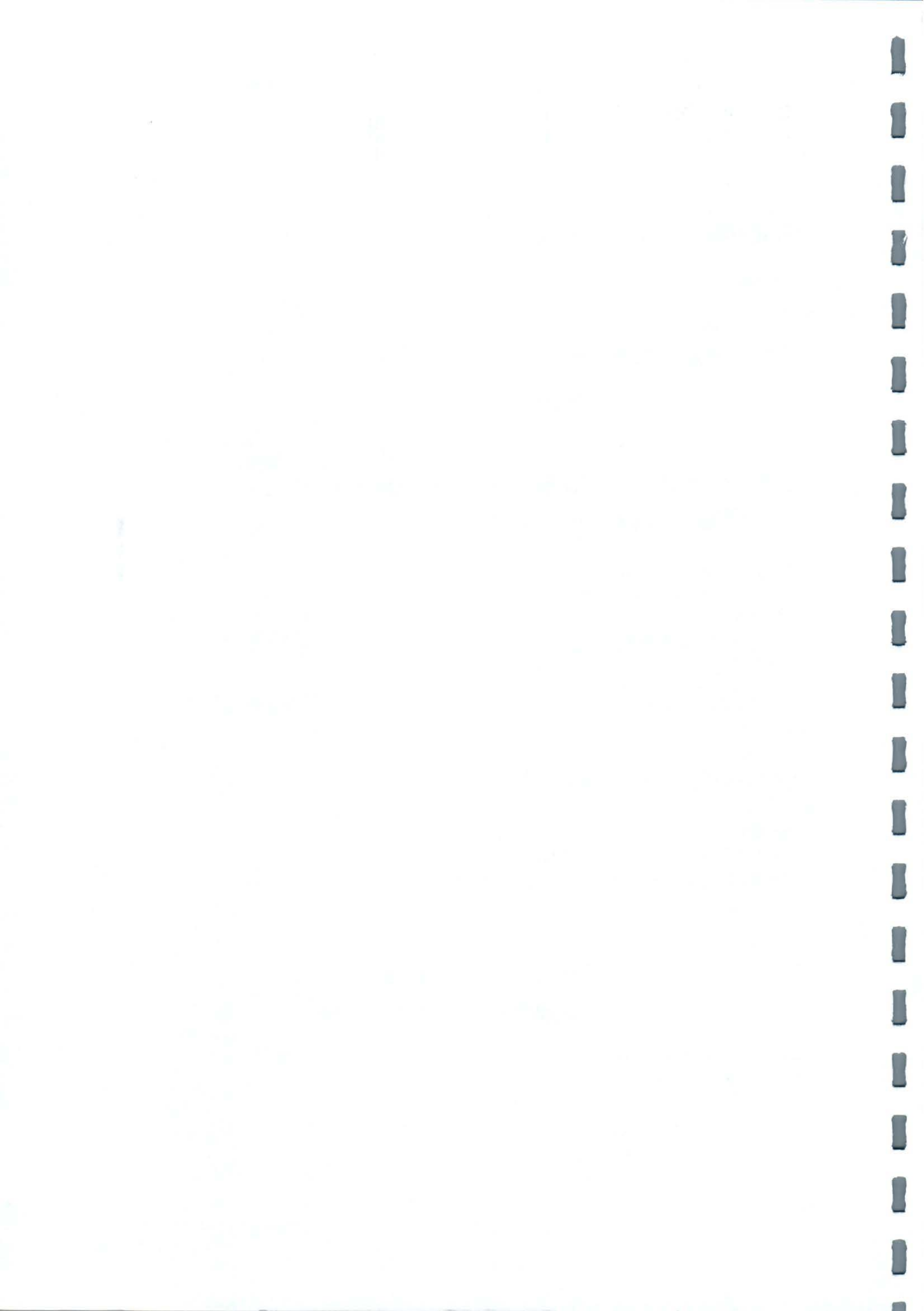
Laboratorio 6.- Applets y Layouts

Applets
Interfaz Gráfica al Usuario (GUI).
Distribución de componentes en la interfaz.

Laboratorio 7.- Entrada/Salida & Manejo de Excepciones

Métodos de la clase File
Manejo de Excepciones
- Excepción IOException

Obtención de datos a partir de archivos en disco.
- Clases :
  - InputStream



Laboratorio 11.- Data Access Builder

Beans

Programación Visual

- Editor de Composición Visual

Data Access Builder

- Utilización de los beans generados para desarrollar aplicaciones

Capítulos (para Clases) :

1. Introducción, Características de Java, Visual Age for Java
2. Tecnología y Conceptos de Orientación a Objetos
3. El lenguaje Java. Tipos de datos simples y estructuras de control
4. El lenguaje Java. Tipos de datos compuestos Modificadores para clases y miembros
5. Interfaces y Clases internas. Bibliotecas de clases.
   El paquete java.lang
6. Clases para Interfaz gráfica al Usuario (GUI) :
   El paquete java.awt , El paquete java.swing
   Programación con eventos : El paquete java.awt.events
7. Applets y Aplicaciones
8. Manejo de Errores y Excepciones
9. Hilos y Sincronización (Multithreading)
10. Beans
11. Programación Visual
12. Bases de Datos
13. JDBC
14. Data Access Builder

Laboratorios

1. El Ambiente de Desarrollo Integrado del Visual Age for Java
2. Tipos simples. Estructuras control
3. Tipos compuestos : Clases. Herencia
4. Tratamiento de cadenas
5. Programación con Eventos
6. Applets y Layout
7. Entrada/Salida & Manejo de Excepciones
8. Hilos y Sincronización
9. Beans y Programación Visual
10. Bases de Datos y JDBC
11. Data Access Builder

Programación Orientada a Objetos con JAVA

Objetivos del curso:

Brindar las bases para una educación en los fundamentos de la tecnología orientada a objetos, la programación orientada a objetos y en particular el desarrollo de aplicaciones y applets para internet utilizando el lenguaje Java.

Lograr que el alumno obtenga la habilidad de diseñar clases para aplicaciones y applets con sólidas bases de paradigmas de objetos y programación por eventos.

Lograr que el alumno obtenga la habilidad de desarrollar e implementar aplicaciones y applets usando lenguaje Java.

Se usa como herramienta didáctica y práctica a uno de los ambientes integrados (IDE) para desarrollo de aplicaciones de misión crítica más completos y poderosos del mercado : el Visual Age for Java de IBM.

Java

Qué es Java ?

Java es un lenguaje de programación orientado a objeto desarrollado por Sun Microsystems y aprovechado para el desarrollo de aplicaciones para la Internet.

Es un lenguaje de propósito general, que puede ser usado para desarrollar cualquier tipo de aplicaciones.

Produce un programa : Seguro, portable, robusto, multihilo, interactivo.

Es más que una alternativa a C++.

Sirve para confeccionar "applets".
Permite convertir un WebSite estático en dinámico e interactivo.

Un applet es una pequeña aplicación que se almacena en un servidor Web (HTTP) para que pueda ser accedida por una estación "cliente" (PC), trasmitida a través de la red (Internet/intranet) y la que se instala automáticamente y es ejecutada por el programa navegador de Web ("browser") como parte de un documento (HTML).

Tiene elementos (aprovecha la experiencia) de C++ y otros lenguajes de programación.

Tiene librerías (clases) altamente especializadas para Internet.

Introducción

Cuando aparece una nueva tecnología, generalmente existen detractores por motivos varios: teóricos, prácticos o simplemente económicos (siendo este último muy común).

Cuando en 1995 aparece Java, se da el fenómeno inverso : todos, hasta los más directos competidores de Sun Microsystems, quien lo lanzó, son parte de un gran entusiasmo y prácticamente de inmediato planifican o comienzan a incorporarlo en sus diversos productos de software.

Pero porqué casi todos apuestan por Java ?

1.- Es un lenguaje de programación orientado a objetos

Para ese entonces ya nadie dudaba que este paradigma de programación (OOP por sus siglas en inglés) es el más adecuado para las necesidades actuales de desarrollo de software complejo y distribuido. Adicionalmente, Java nace corrigiendo muchos de los errores de diseño y peligros de otros lenguajes OOP y sobre todo haciéndolo de una manera SIMPLE.

2.- Es independiente de la plataforma

Permite desarrollar y desplegar aplicaciones con independencia del equipo y sistema operativo donde correrá.

3.- Permite desarrollar aplicaciones para el Internet

El gran éxito del Internet y la ahora totalmente factible posibilidad de usar este ambiente y sus herramientas, para desarrollar aplicaciones internas (Intranet) y de uso externo (Extranet) a las organizaciones fue también uno de los motivos importantes para este entusiasmo. Java permite desarrollar aplicaciones que son cargadas por un navegador de Web y ejecutadas de manera segura en la estación.

Características

Simple y poderoso

Tiene conceptos básicos de programación orientada al objeto, lo que proporciona un paradigma de simplicidad, pero a su vez permite llegar a un nivel lo suficientemente bajo (detalle) como para poder realizar casi cualquier cosa.

Seguro

Los programas de Java no pueden llamar a funciones globales y ganar acceso a recursos del sistema arbitrarios. Entonces se puede ejercer un control sobre los programas ejecutables de Java, lo que no es posible en otros sistemas.

Robusto

Java verifica el código mientras se escribe y una vez más antes de ejecutarlo. Esto permite que el programa se comporte de una manera predecible, bajo diversas condiciones.
Java ayuda a evitar fallas por condiciones excepcionales (división por cero, acceso a archivo no existente, etc.) mediante un manejo de excepciones orientado a objetos integrados.
Java prácticamente elimina la posibilidad de falla en la gestión de memoria (no liberarla o, peor aun, liberar aquella que no corresponde) mediante la facilidad de "recojo de basura".

Interactivo

Java tiene varias características que permiten escribir programas que hacen muchas cosas al instante, sin perder el control de lo que debería suceder y cuándo. El intérprete de Java soluciona esta necesidad de sincronización entre múltiples procesos con "hilos múltiples" (multithreading) de sencilla utilización, que permiten pensar en el comportamiento específico, sin tener que integrarlo en el clásico modelo de programación global de "bucle para el evento".

La historia de Java

1992 :

La compañía Sun Microsystems lleva a cabo el proyecto "Green" para comunicar diversos aparatos o dispositivos electrónicos.

El prototipo era un sistema operativo distribuido "Star7" donde cada dispositivo era parte del todo.

Se crea el lenguaje "Oak", un intérprete para este y clases con este lenguaje para la GUI.

1993 :

(Marzo) FirstPerson, subsidiaria de Sun, responde a una RFP (petición de ofertas) para el proyecto de televisión interactiva de la cia. Time Warner : requerían un pequeño sistema operativo en microcódigo, orientado a objeto, capaz de suministrar flujos de vídeo MPEG sobre redes ATM. Sun pierde este negocio y tampoco llega a un acuerdo con otra cia. de televisión por cable : 3DO.

1994 :

Sun Implementa un browser de Web, al que llamaron "WebRunner" que podía ejecutar clases transferidas desde un servidor HTTP.

La 1ra applet : la mascota "Duke" aparece saludando desde una página.

1995 :

En enero renombran a Oak por Java y a Webrunner por HotJava.
En noviembre aparece en la Web la 1ra versión beta oficial. JDK 1.o

1996:
En diciembre se libera JDK 1.1.

Características del lenguaje Java

JAVA ES MEJOR MÁS POR LO QUE NO TIENE, QUE POR LO QUE TIENE !!

Variables Globales (expuestas a todos)

En Java, el único espacio de nombres global es la jerarquía de clases. No es posible crear una variable global que esté fuera de todas las clases.

GoTo (para hacer tallarines)

Java reserva GoTo como palabra clave para EVITAR que los programadores la utilicen de manera confusa. Mas bien, tiene un concepto de sentencias Break y Continue etiquetadas para cubrir los únicos usos importantes del GoTo.

Punteros (mentirosos y peligrosos)

A pesar que los descriptores de objetos de Java están implementados como punteros, Java no tiene la capacidad de manipular punteros directamente, impidiendo que puedan referenciarse direcciones de memoria de manera arbitraria. Con esto impide lo que otros lenguajes (como C++) permiten : que no existan datos privados de verdad.

Asignación de memoria (buscando un sitio)

En C++ hay que preocuparse por la gestión de la memoria:
malloc() y new() asigna una cantidad de memoria (bytes) devolviendo la dirección del bloque correspondiente. free() y delete() libera un bloque, para que esté disponible por el sistema nuevamente.
En Java cualquier estructura de datos compleja es un objeto que se crea a través del operador new, que le asigna una cantidad de memoria de un "montículo", por lo que el programador no debe preocuparse en una secuencia de direcciones ascendentes. El operador de Java new obtiene, más que una dirección de memoria, un "descriptor" de objeto, de manera que el programador no debe preocuparse por ubicaciones. Incluso la memoria real asignada a un objeto podría moverse en tiempo de ejecución y esto sería transparente. Asimismo existe la facilidad para que el programdor se despreocupe de la devolución de memoria al sistema ya que en Java cuando no existe referencia alguna a un objeto, la memoria que está

acterísticas o de

torno con muchas facilidades

entorno Java provee varias clases necesarias para la interacción de un programa otros.

tectura neutral

do se diseñó Java se tomaron consideraciones , tanto para el lenguaje como el intérprete, de manera que un programa pueda "escribirse una sola vez, cutarse en cualquier sitio, en cualquier momento y para siempre".
ultiplataforma : Solaris, Iris, Linix, HP/UX, OSF, Windows 95, Windows NT, /2, Macintosh.

pretado y de alto rendimiento

va compila a una representación intermedia llamada "código de byte", el que se interpretar en cualquier sistema que tenga un intérprete de Java. Este código te fue diseñado de tal manera que sea sencilla la necesaria traducción a digo de máquina nativo. Con esto se logró que un intérprete sea de alto miento.
parte de un programa Java se reduce a una secuencia de bytes que presentan instrucciones en una "máquina virtual" y no en una máquina esador) específica.

ptimo para Internet

P : Java provee clases para gestionar limpiamente protocolos de Internet. cluye implementaciones de ftp, http, nntp, smtp junto con conectores de red de nivel (sockets) e interfaces de nombrado.
W y HTML : Java está diseñado para cumplir los requisitos de entrega de ntenidos interactivos usando applets insertadas en sus páginas HTML. Las clases as de manejo de cadenas son especialmente adecuadas para el tipo de anejo de texto basado en HTML.
bución por la red : las clases de Java se envían con facilidad e incluso pueden lizarse en el proceso, resolviendo el clásico problema de control de versiones.

Características del lenguaje Java

Tipos de datos frágiles (dependencia del procesador)

Java eligió tamaños adecuados para todos los tipos numéricos de manera de garantizar resultados independientes de la plataforma de hardware. La mayor parte de los lenguajes hacen depender la implementación de los tipos al tamaño natural de "palabra" de la máquina, diferenciándolos así de acuerdo al procesador.

Conversión de tipos (insegura)

Como los descriptores de objetos de Java incluyen información de la clase a la que pertenecen, puede hacerse comprobaciones de compatibilidad de tipos en tiempo de ejecución y controlarlo con excepciones. En C++ no hay forma de detectar esto en tiempo de ejecución, pues los objetos son simplemente punteros a direcciones de memoria. Algo parecido sucede con las listas de argumentos de longitud variable donde la comprobación de tipos queda bajo responsabilidad del programador.

Archivos de cabecera (descabezados)

El lugar natural para la información de una clase es el mismo archivo compilado. En Java no hay archivos de cabecera. El tipo y visibilidad de los miembros de la clase se compilan en el archivo de la clase.
C++, al tener archivos de cabecera (compilados separados que contienen las características o "prototipo" de una clase) permite la posibilidad de tener una o más versiones distintas a la verdadera clase implementada. Como si esto fuera poco, puesto que la interfaz a un programador a una clase compilada es su archivo de cabecera, la clase está tan expuesta como para incluso convertir datos privados en públicos!.

Preprocesador (innecesario)

Los compiladores de C y C++ recurren al preprocesador, que se encarga de buscar comandos especiales (que empiezan con #).
Java no requiere de un preprocesador; dispone de una palabra reservada *final* para declarar constantes (en vez de usar #define).

Java provee

Clases de núcleo
Al estar Java implementado en Java, este lenguaje requiere de ciertas clases para funcionar :
Clases String para manipulación de texto.
Clase Thread para el manejo de "múltiples hilos"
Clases Exception y Error para el manejo de excepciones

Utilidades para applets
Estructuras de datos y métodos asociados para implementación de applets y aplicaciones complejas. Todo implementado como clases.

Entrada/Salida
Los Objetos InputStream y OutputStream son utilizados como un modelo de flujos uniformes de manera que estos son traducidos a cualquier forma específica como un sistema de archivos, una red, un dispositivo de entrada o uno de salida.

Clases para manejo de redes
Java provee clases para trabajar con conectores de red de bajo nivel (sockets) facilitando manejar direcciones de internet así como los flujos de E/S que van hacia y desde estos conectores desde y hacia otras máquinas del Internet.

Clases para manejo de entorno gráfico (GUI)
Clases para manejo de la mayoría de objetos gráficos comunes a todas las plataformas que presentan interfaz gráfica al usuario (GUI)

Comparación con otros lenguajes

Java :

Es el siguiente paso al lenguaje C++.

Soporta librerías de clases como C++.
Es multipropósito como C o C++.

Todo en Java son objetos y clases, no existe el concepto de estructuras de datos.

Soporta la mayoría de propiedades de un lenguaje orientado a objetos.

Permite le creación de pantallas dinámicas.

Existen herramientas como el VisualAge Java que permiten el desarrollo visual de aplicaciones.

Soporta componentes especializados reutilizables : Java Beans, los que son comparables con OCX y ActiveX.

Maneja concurrencia como Mesa.

Controla excepciones como C++.

Enlaza código en forma dinámica como Lisp.

Permite definir interfaces como Objetive C.

Administra en forma automática la asignación dinámica de memoria como Lisp.

Tiene sentencias simples como C.

Genera código compacto como ensamblador.

La Arquitectura Neutral de Java

El problema de la incompatibilidad de plataformas

Un programa funciona sólo en la plataforma para la cual fue diseñado.
Se han normalizado algunos lenguajes de programación (por ejemplo ANSI C) con lo que se mantiene cierta compatibilidad, sin embargo siempre es necesaria la recompilación el código fuente.
Cada fabricante agrega funcionalidad no estándar a un lenguaje de programación.
Cada sistema operativo restringe a los programas en los servicios y capacidades de los mismos.
En general todo programa se tiene que rehacer o adaptar para que funcione en una plataforma diferente para la que se diseñó.

La estrategia de Java

Java fue diseñado para funcionar en sistemas heterogéneos.
Java no depende de la plataforma de hardware/software en la que se ejecuta.
Java provee las mismas librerías (API) para cada sistema operativo en el que se ejecuta.
Java no requiere de recompilar el código, se programa y compila solo una vez.

Java es de arquitectura neutral.
Java trabaja con un código objeto (Byte Code) que representa instrucciones de máquina para un procesador virtual.
Java requiere de un intérprete (JVM por las siglas en inglés de Java Virtual Machine) para cada sistema operativo en el que se ejecuta.

El beneficio de esta arquitectura es la de comprar una vez, se ejecuta en cualquier sitio, se escribe una vez y se vende para cualquier plataforma.

VisualAge for Java

Es un ambiente visual e integrado que soporta el ciclo completo de desarrollo de programas Java.

Características:
Es 100% compatible con Java.
Presenta un ambiente gráfico de desarrollo integrado.
Maneja múltiples proyectos. Administra archivos y controla versiones.
Tiene un compilador incremental.
Permite el desarrollo de aplicaciones corporativas cliente/servidor n-tier.
Permite conexión con servidores corporativos.
Puede ser usado para una amplia gama de proyectos.

Permite el desarrollo de applets y aplicaciones mediante técnicas RAD (Rapid applicaction Development) :

Se diseña la interfaz de la aplicación.
Se define el comportamiento de los elementos de la misma (objetos gráficos)
Se define la relación entre la interfaz y la aplicación.

VisualAge for Java (VAJ) genera el respectivo código en Java de todo lo que fue especificado de manera gráfica e intuitiva.

VAJ provee facilidades para las tareas :
- Creación de un nuevo applet.
- Creación de un elemento de programa (Proyecto, paquete, clase, interfaz o método)

Las versiones:
VAJava Professional combina la programación visual y por eventos con el lenguaje Java en un poderoso IDE (ambiente de desarrollo por sus siglas en inglés) gráfico.

VAJava Enterprise agrega a este IDE, las capacidades para desarrollar aplicaciones escalables y simplifica la labor del desarrollo de aplicaciones cliente/servidor.

Las applets creadas con VAJava correrán en cualquier browser compatible con Java.

Las aplicaciones desarrolladas con VAJava correrán en cualquier plataforma provista de una máquina virtual de Java.

VAJava es una real solución para empresas que quieran extender su alcance a través de la Web y con tecnología independiente de la plataforma.

El ambiente de desarrollo integrado (IDE)

El IDE de VAJava provee facilidades para examinar y manipular el código.

El Workbench es la ventana principal del IDE y provee una variedad de visualizar y manejar los objetos de aplicación.

Los objetos de código son mostrados en una jerarquía de ensamble :

Un Proyecto contiene paquetes
Un Paquete contiene clases e interfaces.
Una Clase contiene métodos y campos
Una Interfaz contiene métodos y campos estáticos (final static fields)
Un Método es una construcción de lenguaje Java que especifica cierta lógica o función.

VAJava permite :

Construir programas Java interactivamente.
Correr programas Java.
Correr fragmentos antes de incluirlos en clases.
Probar y corregir errores mientras se codifica el programa.
Manejar varias ediciones y versiones de programas.
Importar programas fuente y bytecode desde el sistema de archivos del ambiente.
Exportar programas fuente y bytecode hacia el sistema de archivos del ambiente
Construir, modificar y usar Java beans.

El Workbench

A través de las "pestañas" del workbench puede irse introduciendo en el detalle de los objetos.

A través de la pestaña de errores puede verse errores de compilación y accederse automáticamente el código fuente correspondiente.

Los browsers

El ícono de la esquina superior izquierda de la ventana identifica al navegador en uso.

El navegador de proyectos permite trabajar con todos los elementos de un programa particular, los que pueden accederse - mediante las pestañas respectivas - por paquete, clase, interfaz o edición.

El navegador de paquetes permite manipular las clases e interfaces de un paquete en particular.

El navegador de clases permite manipular código a este nivel (de clases).
Para clases que provean funcionalidad para interactuar con la interfaz gráfica de usuario, contempla en una de sus pestañas al Editor de composición, que permite programación visual.

El editor de composición visual

Permite diseñar la interfaz al usuario de la aplicación muy rápidamente.

Además de permitir crear y modificar componentes visuales, permite a su vez conectar estos objetos gráficos con otros objetos no visuales que provean la funcionalidad requerida por la aplicación.

Los componentes visuales (objetos gráficos) son sencillamente "tomados" de una paleta y "soltados" en la ventana que los contendrá.

Usando la barra de herramientas puede cambiarse el tamaño, disposición y alinearse los componentes (objetos gráficos).

También se dispone de una hoja de propiedades que permite actualizar las de cada objeto contenido en la interfaz diseñándose.

La gran potencia de esta herramienta es que permite definir, también de una manera visual, las relaciones funcionales entre componentes gráficos y entre éstos y componentes no visuales. Esto se realiza mediante conexiones entre objetos. estas conexiones estarán representadas por flechas.

Una vez diseñada la interfaz y conectados los componentes, podrá guardarse o probarse. VAJava generará todo el código.

VAJava usa tecnología de JavaBeans. Esto permite diseñar beans para GUI así como beans para lógica de negocio.
Al usar VAJava el modelo de componentes estándar en la industria, se permite importar cualquier clase de Java al IDE de VAJava. Una vez importado un bean puede agregarse funcionalidad "especializada" o "adaptada" a la realidad de su organización.

El editor de composición visual permite crear de manera visual componentes NO VISUALES !

Facilidades para compilación, pruebas y control de versiones

VAJava permite seguimiento del código a través de breakpoints.

Ofrece compilación dinámica e incremental :
Cada vez que hay una modificación, sólo se compila el elemento actualizado.

Es más, puede actualizarse código mientras este mismo está corriendo. La ejecución "se percatará" de este evento y seguirá corriendo tomando en cuenta la actualización.

Se ofrece registrar ediciones abiertas y ediciones en versión (históricas).

Puede colocarse versiones a proyectos, paquetes y clases. Al colocar una versión, todos los componentes de este serán registrados con la misma versión.

VAJava provee una ventana para editar y probar código temporal de manera de no tener que definir clases sino hasta que la idea o pruebas hayan sido completadas.

Con esta ventana "scrapbook" y con la ventana de consola, se tiene prácticamente un microambiente para pruebas rápidas y verificaciones pequeñas.

Aplicaciones escalables y de misión crítica

Adicionalmente al visual y poderoso ambiente de desarrollo, la edición ENTERPRISE de VAJava ofrece un conjunto integrado de herramientas para desarrollar aplicaciones de misión crítica y que puedan ser escalables horizontal y verticalmente :

El Enterprise Access Builder permite extender el alcance de las aplicaciones Java para acceder bases de datos relacionales, transacciones CICS y aplicaciones de C++.

Asimismo permite distribuir las aplicaciones a través de la red con una variedad de protocolos.

Las partes del Enterprise Access Builder :

CICS Access Builder

Genera código que puede integrarse en los programas de manera de acceder transacciones CICS.

RMI Access Builder

Distribuye código entre cliente y servidor, en un ambiente distribuido.

C++ Access Builder

Genera código que puede integrarse en los programas de manera que puedan hacerse llamadas a aplicaciones C++.

Data Access Builder

Genera código que puede integrarse en los programas de manera de acceder datos en Bases de datos.

La industria del Software

La crisis :

- Los tiempos en desarrollo de nuevo software o para cambios, son demasiado extensos y las empresas no están dispuestas a esperar.
- La frecuencia de fallas cada vez que se realizan cambios significativos, es muy alta.
- Altos costos de producción y mantenimiento de sistemas.
- Muchos proyectos son abandonados antes de finalizar.
- Pérdida de ventajas competitivas por la incapacidad de adaptarse rápidamente a los cambios del mercado.

En la industria del software nunca existió realmente metodologías para "industrializar" la producción de software.

Uno de los factores principales en la velocidad del desarrollo de la industria del Hardware ha sido la existencia de componentes estándares, intercambiables y reutilizables.

Evolución de la Programación

La "prehistoria"

Recursos (memoria) limitados

Programación en lenguaje máquina

La "humanización"

Lenguajes de programación

1era aproximación para industrialización: el enfoque descendente (top-down)

Descendente en los niveles de abstracción :
Primero se estudia el problema/sistema en términos generales (alto nivel) para luego ir "bajando" de nivel a aquellos de mayor detalle.
En programación : se desarrolla el nivel más abstracto y se diseñan "rutinas" para los detalles de implementación más específicos.
Se recomendaba tener en cuenta la modularidad y cohesión.
Se recomendaba evitar variables globales y mas bien pasar punteros a estructuras.

La aproximación real

Herramientas : Lenguajes de "4ta generación", entornos de desarrollo, CASE.

El cambio de paradigma (pero aprovechando lo anterior)

Programación Orientada a Objetos

Tecnología Orientada a Objetos

Los Paradigmas:

Objetivismo
Resolución de problemas complejos enfocándolos a situaciones de objetos de la realidad.

Comunicación por mensajes
Los humanos, animales y máquinas se comunican a través de mensajes.

Polimorfismo
En el mundo real un objeto suele presentar muchas formas, no sólo físicas sino de comportamiento.

Generalización / Especialización
En el universo prácticamente todo presenta jerarquías. Los objetos pueden agruparse por sus similitudes pero van diferenciándose por características especiales, conformando jerarquías de clases.

Herencia
Los objetos especializados de cierta clase presentan las mismas características que la clase superior a la que pertenecen o "heredan" estas características, tanto físicas como de comportamiento.

Ocultamiento
Resolución de problemas complejos, ocultando la complejidad en niveles altos de abstracción, de manera que sólo se vea la interacción del sistema (producto o solución del problema) con su entorno.

Encapsulamiento
La "caja negra" permite ocultar la complejidad, pero el encapsulamiento, además, protege a los componentes de otros, los que sólo conocerán lo que el componente les permita en su interfaz exterior.

Componentes reutilizables
Al sólo conocerse la interfaz (cómo solicitar) y la función de un componente, es más fácil su reutilización.

Contenedores (Ensambles)
Al tener identificados componentes claramente funcionales y reutilizables, pueden resolverse problemas de estandarización, intercambiabilidad, producción voluminosa y ensambles.

Programación Orientada a Objetos (OOP)

Algunas áreas donde se aplica la tecnología de objetos :

Lenguajes de Programación
Bases de datos
Dibujo y Visualización
Sistemas Operativos
Interfaz de usuario
Metodología de análisis y diseño
Sistemas de conmutación telefónica

En los lenguajes de programación en particular y en el desarrollo de software en general, estos paradigmas llevan a una nueva forma de desarrollar :

Una orientación al producto (software) en vez de al proceso de desarrollo mismo.
Un cambio hacia la eficacia con eficiencia.

Debe lograrse el control del nivel de integración de los componentes de software aproximando su uso inclusive al consumidor final (quien requiere por ejemplo objetos de negocio).

Historia (Resumida) de la OOP

1967 : SIMULA
197? : SmallTalk
1980 : C++
1995 : Java

En la diapositiva puede apreciarse las ventajas de la OOP.

El encapsulamiento permite :

1.- Uso de funcionalidad de manera segura y no compleja
El encapsulamiento protege a los datos del objeto de otros objetos, así como les oculta los detalles de implementación del objeto.

2.- Modularidad
Puede modificarse las partes internas (datos y métodos privados) de un objeto sin afectar (ni preocuparse por ello) al resto de la aplicación.

Clase y Objeto (instancia de clase)

Clase

Una clase es una generalización de características de objetos.

Existen dos tipos de características :

Atributos : información de identificación o descriptiva o de estado.

Comportamiento : funcionalidad o conducta ante solicitudes.

En OOP:

Una clase hace referencia a los objetos que presentan los mismos tipos de datos y métodos para manejarlos y comunicarse con otros objetos.
Clase es una abstracción que representa un conjunto de objetos de igual estructura (datos) y comportamiento (métodos).

Objeto

Instancia particular de una clase.
Tiene las características que los de su misma clase pero se distingue por los valores de estas características.

En OOP:

Un objeto es una instancia creada a partir de su clase y ocupa espacio (memoria) y tiene una duración : se crea y puede dejar de existir (se destruye).
Se distingue de otros objetos de su misma clase por los valores de sus datos que reflejan atributos y estados del comportamiento.

Un Objeto es una instancia única (identificable) que mantiene la estructura y comportamiento definidos por la Clase, como si ésta última hubiera servido de molde o plantilla. Es por este motivo que los objetos también son llamados instancias de clase.

En todos los lenguajes OO, las cada clase tiene un método por el cual pueden crear ("a su imagen y semejanza") instancias (objetos de esa clase).

Persistencia

En el mundo real :

Por ejemplo una persona ("objeto del mundo real") nace (técnicamente se podría decir que es creado por sus padres) y muere (técnicamente se podría decir que funcionalmente es destruido).

Asimismo puede cambiar de ubicación y sigue siendo la misma persona (trasciende el espacio).

Cuando mueren los padres de una persona, ésta sigue existiendo (trasciende la muerte de su creador).

En OOP:

Un objeto, que es una instancia creada a partir de su clase, ocupa espacio (memoria) y a pesar que cambie de ubicación de memoria seguirá siendo el mismo objeto.

Entonces trasciende el espacio.

También tiene una duración : se crea y puede dejar de existir (se destruye).

Asimismo trasciende el tiempo pues sigue existiendo a pesar que su clase ya no exista más.

Encapsulamiento

En el mundo real :

Un objeto no exhibe toda su información ni muestra todos los detalles de su funcionamiento.

En un programa OO, tanto los datos (su declaración y almacenamiento) como el código (que especifica la lógica) que los manipula son "encapsulados" en una Clase que define el comportamiento y protege a los datos y código de otro código que trate de acceder a éstos de manera arbitraria.

Además de esta importante ventaja de seguridad, el objeto de software provee una interfaz de mensajes a través de la cual puede solicitársele haga algo, lo que permite que otros objetos de software lo utilicen sin necesidad de conocer los detalles de implementación.

Los objetos encapsulan datos que sólo pueden accesarse a través de métodos que el consumidor de software debe invocar (con el protocolo o conjunto de mensajes correspondientes) para obtener un servicio específico ("funcionalidad"). Entonces el consumidor debe conocer QUE hace cada componente, dejando a nivel "interno" el COMO. De esta forma, ante un cambio, es mucho más fácil identificar la ausencia de un componente.

Los objetos pueden simular entes del mundo real implementándose como bloques de software que encapsulan tanto datos como procesos.

Un objeto puede contener otros objetos.

Quien desarrolla puede hacer cambios a la implementación de una clase, sin que esto lo noten los usuarios de la misma.

Herencia

Es la abstracción o más precisamente el paralelo con las relaciones jerárquicas entre objetos del mundo real, donde ciertos objetos tienen las características (atributos y comportamiento) generales de otros (superclases) pero adicionalmente características especiales sólo de ellos mismos (de las subclases).

En el contexto de objetos de software, ya se ha mencionado que cada Clase encapsula datos y métodos, entonces cualquier subclase tendrá las mismas características MAS cualquiera otra como parte de su especialización (que precisamente la haga especial).

Los objetos de una "clase" heredan características (datos y/o métodos) de clases más generales.

Ante un cambio, es fácil identificar la necesidad de "especializar" o particularizar una clase existente o "Generalizar" una clase para una mayor cobertura de utilización (y reutilización).

En OOP la herencia está siempre implícita cuando se define una clase en base a otra. En este caso, la primera será una subclase de la segunda (que será su superclase).

Una subclase puede:

Añadir propiedades (datos) y funcionalidad (métodos) adicionales.
Modificar métodos heredados.
Incluso desactivar algunos métodos heredados.

Una clase puede ser tan abstracta que incluso nunca genere objetos (serán sus subclases las que los generen).

Polimorfismo

Polimorfismo se traduce como : un objeto y muchas formas.

Es la abstracción o similitud con el caso de la realidad donde un objeto reacciona ante un mismo mensaje de diversas formas.

En la realidad puede observarse polimorfismo en dos circunstancias:

1.- Polimorfismo en la misma clase
2.- Polimorfismo en las subclases de una clase

En OOP se simula ambos casos.

1.- Polimorfismo en la misma clase

Un objeto de la realidad puede reaccionar (comportarse) a un mismo mensaje de acuerdo al tipo de información contenido:

En OOP se denomina

Sobrecarga estática de métodos

Un objeto de una clase puede reaccionar a un mismo mensaje de acuerdo a éste último (al tipo de dato, que es un objeto, en el mensaje).

Por lo tanto en una misma clase puede existir más de una implementación de un mismo método. Las diferentes implementaciones del método tendrán el mismo nombre pero no la misma especificación.

Polimorfismo

2.- Polimorfismo en las subclases

En la realidad, objetos de la misma clase pueden reaccionar ante un mismo mensaje de manera diferente sólo por el hecho de pertenecer a subclases diferentes.

En OOP se denomina:

Polimorfismo dinámico por niveles (sobreescritura de métodos en las subclases)

Usando subclases puede construirse por niveles un polimorfismo dinámico y en tiempo de ejecución, cubriendo la falta de flexibilidad anterior (sobrecarga estática de métodos).

Aunque no se tenga el fuente, siempre podrá implementarse una subclase con un método cambiado o especializado (se sobreescribe el método).

Cualquier objeto que llame al método, ni siquiera tiene que saber qué subclase ejecutará el método, simplemente deberá invocarlo.

Comunicación por mensajes

Los objetos de la realidad interactúan a través de la comunicación.

El envío y recepción de mensajes que :

Pueden no ser entendidos por el receptor

Pueden ser entendidos pero no atendidos

Pueden ser entendidos y atendidos

Para que se establezca comunicación entre dos objetos de la realidad, se requiere un protocolo (conjunto de mensajes) que sea conocido por ambos.

En OOP

La comunicación entre objetos de software también se da a través de mensajes.

El objeto emisor (más conocido como "cliente") solicita algo (a veces llamado "servicio") a un objeto "servidor" quien provee dicho servicio a través del método que se invocó en el mensaje.

El mensaje es el nombre del método y la especificación del servicio se da a través de los parámetros pasados al método.

El objeto cliente debe conocer la "manera correcta" de solicitar el servicio : especificando los tipos de datos correspondientes a los parámetros del método en cuestión.

Cómo definir Objetos al diseñar software ?

Un objeto es definido en base a:

Su comportamiento:
 Responde a la pregunta ¿qué hace?.
 Implementado en software como funciones.

Sus atributos:
 Responde a la pregunta ¿qué sabe?.
 Implementado en software como variables (otros objetos).

Un objeto puede:
 Cambiar de estado
 Manifestar ciertos comportamientos
 Ser manipulado por distintos estímulos o eventos
 Existir en relación a otros objetos

Candidatos a objetos al modelar sistemas:
Cosas tangibles, roles , lugares, organizaciones.

No candidatos :
Emociones, atributos (como color, belleza, etc.), entidades que normalmente son objetos pero que, para un determinado modelo son atributos.

El lenguaje Java

Al ser Java un lenguaje basado en el paradigma de objetos, sus programas son implementados como clases.

Incluso el nombre de la clase es usado por el compilador para el nombre de la clase ejecutable (archivo ya compilado).

Las personas que conocen el lenguaje C o C++ apreciarán que la sintaxis es muy parecida. En el ejemplo puede apreciarse los paréntesis para colocar los parámetros de las funciones (en este caso métodos), así como el punto y coma para indicar el fin de una sentencia.

public es un especificador de acceso. También se verá que existen otros como **private** (privado) y **protected** (protegido).

main siempre debe declararse static pues cuando empieza un programa no existen objetos (instancias creadas).
Los métodos estáticos sólo pueden referirse a variables estáticas e invocar métodos a su vez estáticos.

Las variables y métodos estáticos son más conocidos como variables de clase y métodos de clase.

Identificadores

Los identificadores sirven, como su nombre lo indica para identificar variables, métodos o clases.

Java es "sensible a las mayúsculas y minúsculas". Por lo tanto bastará una diferencia de este tipo para que el compilador reconozca un nombre como un diferente objeto.

Se recomienda :

No usar los caracteres subraya ni dólar para el primer caracter, de manera de poder usar librerías de C/C++ sin temor de "cruzarse" con alguna de sus variables utilizadas, las que precisamente suelen comenzar con alguno de estos dos caracteres.

Usar mayúsculas al inicio del identificador y si lo conforman varias palabras, para dar mayor significado al objeto, también colocar mayúsculas en las primeras letras de las mismas.

Ejemplos : MiPrimeraClase, VectorTemporal.

Identificadores reservados (Keywords)

Los identificadores reservados son de uso exclusivo del sistema Java, por lo que no puedes ser utilizados como identificadores. También se les conoce como "palabras clave".

Java tiene una mayor cantidad de palabras reservadas que C/C++.

Literales (Constantes)

Un valor constante en Java se crea utilizando una representación literal de él. Puede ser un número, un caracter o una cadena de caracteres.

Literales Enteras
Pueden expresarse en :
Formato decimal (base 10). Ninguna notación en especial. Ej. 534
Formato hexadecimal (base 16). Número precedido de 0x. Ej. 0xffff vale 655535.
Formato octal (base 8). Número precedido de 0. Ej. 014 vale 12.
Por omisión las literales enteras se guardan como tipo **int**, que es un valor binario de 32 bits que incluye signo.
Si se desea números mayores a 2**16 o menores que -2**16 (aproximadamente 2'000,000,000) puede usarse la extensión L (o l minúscula) al final del número y en cualesquiera de las bases disponibles, lo que indicará un tipo **long**, a almacenarse en 64 bits. Ejemplos : 69L,   0x7f5a9b65L

Literales de Punto Flotante
Representan valores decimales que tienen fracción.
Pueden expresarse en :
Notación estándar : dígitos seguidos de un punto y más dígitos para la fracción.
Notación científica : número en notación estándar con una notación adicional para indicar la potencia de 10 a la que hay que elevar el número anterior. Esta potencia o exponente se indica con una E seguida por un entero con sin signo.
Por omisión se guardan como tipo **double**, almacenándose en 64 bits y aunque no es necesario, puede redundarse la especificación agregando una D (o d) al final.
Para usar el tipo **float**, de 32 bits de tamaño, debe especificarse una F (o f) al final.

Literales booleanos
Representan los valores lógicos de verdadero y falso. Java proporciona las reservadas **true** y **false** para este efecto.

Literales de un caracter
Representan índices en el conjunto de caracteres Unicode. Son valores de 16 bits que pueden convertirse a enteros y operarse como tales. Los caracteres ASCII (visibles en los teclados) pueden expresarse directamente entre dos comillas simples, pero otros caracteres como los dependientes del idioma deben representarse por su valor Unicode en representación octal o hexadecimal (ver tabla de caracteres especiales en la diapositiva).
Ejemplo : 'H' = \u0048 = \110 ya que ocupa la posición 72 en la tabla Unicode.

Literales de cadenas de caracteres
Representan un texto y se expresan entre comillas dobles. Deben terminar en la misma línea de programa donde comenzaron. En realidad el compilador las

implementa como instancias de la clase String con el valor textual encontrado, aunque esto es transparente al programador.

Operadores

Los operadores especifican una operación o evaluación sobre uno o más objetos (datos u objetos) para generar un resultado.

Aritméticos y de conversión

+ - * / Suma, Resta, Multiplicación, División
% Módulo
++ Incremento en 1
-- Decremento en 1
- Cambio de signo
~ Complemento al bit
& Y binario (AND al bit)
| O inclusivo binario (OR al bit)
^ O exclusivo binario (XOR al bit)
<< Desplazamiento de bits a la izquierda
>> Desplazamiento de bits a la derecha
>>> Desplazamiento de bits a la derecha (sin considerar signo)

De Comparación

== Igual
!= Diferente
< Menor
<= Menor o igual
> Mayor
>= Mayor o igual

Lógicos

& Y lógico (AND) de evaluación completa
| O lógico inclusivo (OR) de evaluación completa
^ O lógico exclusivo (XOR)
&& Y lógico (AND) de evaluación "suficiente"
|| O lógico inclusivo (OR) de evaluación "suficiente"
! Negación lógica
== Igual
!= Diferente
?: Condicional ternario (sintaxis ExpBooleana ? ExpSiTrue : ExpSiFalse

De Conversión

= Asignación
+= Suma y asigna
-= Resta y asigna
*= Multiplica y asigna
/= Divide y asigna
%= Módulo y asigna
&= Y lógico (AND) y asigna
|= O lógico inclusivo (OR) y asigna

^= O lógico exclusivo (XOR) y asigna

Separadores

{ }

Las llaves sirven para contener los valores de matrices inicializadas automáticamente y para definir bloques de código para clases, métodos y ámbitos locales.

;

El punto y coma separa sentencias

,

La coma separa identificadores consecutivos en declaraciones de variables, también items de una lista .

Comentarios

Adicionalmente al clásico comentario de la forma /* comentario */ también se dispone de // para agregar comentarios a partir de un lugar hasta el final de la línea de código.

Asimismo puede colocarse comentarios de la forma :

/** comentario */

para poder hacer uso del documentador "javadoc".

Tipos de datos y declaración de variables

Los tipos primitivos (simples o básicos) son :

Tipos de datos enteros

- **byte** 8 bits
- **short** 16 bits
- **int** 32 bits
- **long** 64 bits

Tipos de datos de punto flotante

- **float** 32 bits
- **double** 64 bits

Tipo de dato lógico

- **boolean**

Tipo de dato caracter

- **char** 16 bits (entero sin signo, que sirve de índice para tabla Unicode)

Conversiones de tipos (Casting)

El casting es generalmente necesario cuando una función retorna un tipo diferente al que se necesita para realizar una operación.
Se especifica colocando el tipo requerido entre paréntesis y a la izquierda del valor a ser convertido.
Ejemplo : char c = (char) System.in.read();

Conversiones sin posibilidad de pérdida de información :

byte	a	short, char, int, long, float, double
short	a	int, long, float, double
char	a	int, long, float, double
int	a	long, float, double
long	a	float, double
float	a	double

Arreglos

Un arreglo es un tipo compuesto (o una construcción) de variables del mismo tipo. Estas variables pueden ser a su vez de tipo simple o compuesto y se referenciarán con el mismo identificador.
Un arreglo puede ser de una o más dimensiones (multidimensionales que en realidad son matrices de matrices) y cada dimensión se especifica con corchetes seguidos al tipo de datos o seguidos al identificador.

Ejemplos :

```
int numeros[ ];
char [ ] letras;
```

A diferencia de otros lenguajes, en Java no se permite especificar el tamaño de un arreglo vacío directamente. Para esto debe usarse el operador new o asignarle una lista de items de su tipo.

Ejemplo :

```
char CaracterAlfabetico [ ] = new char [27] ;
int Digitos = { 0,1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

Cadenas (Strings)

En Java las cadenas son manejadas por la clase String. Incluso los literales tipo cadena son en realidad instancias de la clase String.

Bloques o Ámbitos

En Java , el código se divide en bloques delimitados por llaves.
Un bloque es un conjunto de sentencias entre llaves.
Un bloque puede contener a otros bloques.

Alcance y Visibilidad

Las variables existen desde su definición hasta el final del bloque donde fueron declaradas y no pueden ser invocadas en el exterior de este ámbito.
Una variable declarada en un ámbito sigue existiendo en los ámbitos internos (sub-bloques), desde donde es "visible", es decir puede invocarse, acceder a su valor y modificarse.

Por el motivo anterior, y a diferencia de otros lenguajes, una variable no puede llevar el mismo nombre (identificador) que una de un ámbito exterior al que pertenezca.

Las variables declaradas en ámbitos internos no son visibles ni siquiera en el bloque exterior al que pertenece.

Las sentencias se ejecutan en el mismo orden en que son especificadas en el bloque. Un bloque es tratado como una sentencia. La ejecución de bloques también es secuencial.

No hay límite para la creación y anidamiento de bloques.

if

La sentencia **if** ejecuta una sentencia o bloque (de varias sentencias) en caso que la expresión lógica especificada resulte el valor true.

if - else

Si la cláusula **else** es especificada, se ejecuta la sentencia o bloque seguida a ésta en caso que la expresión lógica de la sentencia **if** resulte false.

Pueden anidarse sentencias if (else if). En estos casos se recomienda la "sangría" o "indentación".

En caso de requerir un anidamiento considerable, es preferible el uso de la sentencia switch.

switch

La sentencia switch permite ejecutar una lista de sentencias de acuerdo al resultado de una expresión.

Si no se coloca la cláusula break, el flujo sigue "cayendo" a las demás listas de sentencias.

Si es colocada la cláusula default, su lista de sentencias se ejecutará en caso de no cumplirse alguno de los valores case.

switch

La ejecución de la sentencia switch es más eficiente que la misma lógica programada a través de encadenamiento de sentencias if-else.

Sin embargo es un error común el olvidar que el flujo sigue "cayendo" una vez ejecutado el case que resultó verdadero (olvidar de colocar un break).
Por este motivo es casi una convención que cuando no hay breaks, se el colocan comentarios "continúa" que informan que no es un error de omisión.

for

```
for (ExpDeInicializacion; ExpLogicaParaSeguir; ExpDeIteracion)
  sentencia; o bloque
```

Ejemplo (clásico) :

```
for (int i = 1; i <= n; i++) System.out.println(i);
```

while

```
while (ExpLogicaParaSeguir)
  sentencia; o bloque
```

do-while

```
do
  sentencia; o bloque
while (ExpLogicaParaSeguir)
```

break (bifurcación incondicional)

La sentencia break hace que el control pase fuera del bucle (a la siguiente sentencia o bloque).

También puede usarse para pasar el control fuera de un ámbito etiquetado :

```
etiqueta1: {
etiqueta2:   {
etiqueta3:     {
                 // ámbito de la etiqueta3
                 if (true)
                   break etiqueta2;
                 // sentencias aquí nunca se ejecutarían
               }
               // sentencias aquí nunca se ejecutarían
             }
             // el control pasa aquí luego del break
           }
```

Nota.- si el if no fuese colocado, el compilador detectaría las líneas de código que nunca se ejecutarían.

continue (bifurcación incondicional)

La sentencia continue hace que el flujo no continue la iteración en proceso dentro de un bucle, pero sin salirse de éste.

En un while o Do-While el control saltará a la condición (a evaluar la "expresión lógica para seguir") y de acuerdo al resultado se continuará o terminará el bucle.

En un for, la tercera cláusula (la expresión de iteración) se ejecuta después del continue.

return

La sentencia return hace que el control pase al objeto que invocó al método que contiene a la sentencia.

Clases y Objetos

Los objetos de software pueden servir para representar objetos del mundo real así como para modelar objetos abstractos para resolución de problemas.
Ejemplo :
La clase "Carro"

En el mundo real se tiene la clase de objeto "Carro", que generaliza a las máquinas que presentan ciertos atributos como :

placa
color
cantidad de puertas
velocidad máxima
etc.

presentan ciertos estados como :

encendido/apagado
velocidad
temperatura del motor
etc.

y además presentan un comportamiento común :

se aceleran
frenan
reciben combustible
etc.

Cuando un objeto del mundo real es modelado como un objeto de software sus propiedades son descritas por **variables de instancia** y su comportamiento es especificado mediante **métodos**.

Cada objeto creado mantiene un juego de variables de instancia con un único conjunto de valores en cada momento. Sin embargo todos los objetos comparten la misma implementación de los métodos de la clase.

Variables de Instancia

Cada objeto creado mantiene un juego de **variables de instancia** con un único conjunto de valores en cada momento.

Las variables de instancia se declaran dentro del bloque principal (entre las llaves de apertura y cierre de la declaración de la clase).

En la jerarquía de objetos que VAJava ofrece, no se ven directamente las variables de instancia sino como parte de la clase, ya que precisamente ésta las encapsula. No se ven en la jerarquía pero si en la declaración (el detalle).

Los métodos son presentados como componentes de la clase y esta dependencia de "estar contenidos" puede apreciarse en la jerarquía del Workbench.

Las variables de instancia no sólo se pueden declarar como cualquiera de los tipos primitivos - tal como se aprecia en la diapositiva - sino también como tipos de clase.

Métodos

La clase "Conductor"

En el mundo real se tiene la clase de objeto "conductor", que generaliza a las personas que - se supone - saben usar objetos de la clase "Carro" y presentan atributos como nombre y comportamientos como presionar el acelerador.

Nótese que al método para implementar el comportamiento "presionar el acelerador" se le ha nombrado (asignado un identificador) igual que al método respectivo en la clase "Carro".

No existe problema en usar el mismo nombre pues como se verá un poco más adelante, para ejecutar un método es necesario hacer referencia al objeto al que se está "pasando el mensaje" (o solicitando el servicio).

Declaración de métodos

```
TipoDeRetorno IdentificadorDeMetodo (parametros) {
  cuerpo-del-método
}
```

Invocación de métodos (comunicación por mensajes)

Otros objetos acceden a la funcionalidad de un objeto a través de los métodos que éste tiene "visibles". A este conjunto de métodos que pueden ser invocados por otros objetos suele denominarse **interfaz** y a esta invocación de método también se conoce como **comunicación por mensajes**.

Para invocar a un método de un objeto, otro objeto, desde un método, tiene que especificar :

a qué objeto pertenece el método;
qué método es el que se invocará;
los parámetros requeridos por el método.

Todo esto conforma el "mensaje" que el objeto transmisor (o cliente) envía al objeto receptor (o servidor).

Ejemplo :

Un conductor (objeto) para acelerar un carro (otro objeto) sólo presiona el acelerador, sin preocuparse del cómo el auto hace para aumentar su velocidad ni qué mecanismos internos emplea.

Puede observar que el conductor "Hugo" usa **SU** método ***acelera*** (que él conoce) y que consiste en presionar el acelerador con una cierta presión. Este método a su vez envía el mensaje al objeto carro "IG1524" : "pisé **tu** acelerador con presión 2" o lo que es lo mismo : invoco **TU** método ***acelera*** (esta vez el método del carro) con parámetro "2".

Declaración de clases (Tipos compuestos)

Sintaxis:

```
class IdentificadorDeClase {
  cuerpo-de-la-clase
}
```

Si se quisiera ver desde el punto de vista de compilación:

Cuando se usa la palabra reservada **class**, el identificador usado se registra como nuevo tipo de datos (compuesto). La creación de variables de este tipo es similar que para variables de tipos primitivos : Para crear una **variable de referencia** a una clase se especifica el identificador de la clase seguido del identificador para la variable de referencia.

Sintaxis:

```
IdentificadorDeClase   IdentificadorVariableReferencia;
```

Creación de instancias

Para crear un objeto de una clase y tener una variable de referencia asociada al objeto se usa el operador **new**

Sintaxis:

```
IdentificadorDeClase  IdentificadorVariableReferencia = new IdentificadorDeClase();
```

Con esto se está reservando la memoria necesaria para las propiedades (variables de instancia) del objeto. Así es como se crea o construye una instancia.

En la diapositiva se declara el tipo Circulo. Se declaran tres variables de referencia de este tipo y en las mismas sentencias se están asignando a tres objetos nuevos creados con el operador new.

El método constructor

El constructor de una clase es un método que reserva recursos para las variables de instancia del objeto que se esta creando. Tiene el mismo nombre que la clase a la que pertenece. Es invocado cuando se crea un objeto con el operador new, por lo que generalmente su función es la de inicialización de variables.

Puede tener varias implementaciones.

En el ejemplo se tienen dos métodos constructores :
Circulo() y Circulo(int), donde este último requiere un parámetro: el radio del círculo que se quiere crear.

Un método constructor no incluye un tipo de retorno, debido a que nunca devuelven valor.

Los métodos constructores no pueden ser invocados directamente como cualquier otro método, sino siempre cuando se esté creando un nuevo objeto. Al usar el operador **new** para crear objetos, los valores usados como parámetros deben coincidir con los tipos definidos en el correspondiente método constructor.

Puede observarse que las variables de referencia pueden apuntar a diferentes objetos en forma dinámica. Estos objetos también pueden ser parte de una lista de parámetros de algún método. En estos casos estos objetos se pasarán por referencia, a diferencia que las variables de tipos primitivos, que se manejan por valor.

En la diapositiva: mi3ercirculo en un momento estará apuntando al 3er objeto creado del tipo Circulo pero que en el siguiente instante pasará a hacer referencia al 2do objeto creado.

El espacio de memoria dejado por este 3er objeto, al no ser referenciado, será recuperado por el recolector automático de basura de Java. Cuando no hay referencias a un objeto, éste queda en estado para ser destruido automáticamente.

El operador .

Cuando se declara una variable de referencia (de algún tipo compuesto) sin asignarle un objeto específico, ésta apuntará a **null** (ninguno).

Esta asignación también puede especificarse en forma explícita, tal como puede apreciarse en la diapositiva.

Entonces, el operador null puede usarse en la declaración de una variable, a la que no desea asignarse objeto o para "desasignar" una variable que lo estaba de un objeto.

Por ejemplo :

```
Circulo mi7mocirculo = new Circulo(8);
...
mi7mocirculo = null;
```

Referencia a variables de instancia

IdentificadorDeObjeto.VariableDeInstancia

Referencia a métodos (Invocación de métodos)

IdentificadorDeObjeto.Metodo

Herencia

Declaración de clases y derivación de subclases

```
class IdentificadorDeClase extends IdentificadorDeSuperclase {
  cuerpo-de-la-clase
}
```

La subclase así definida "hereda" las variables (propiedades) y métodos (funciones) de la superclase que "extiende" o de la cual deriva.

La real potencia de la herencia no es sólo el hecho de adquisición de propiedades y métodos de la superclase, sino la facilidad de "especializar" la subclase con propiedades y métodos adicionales.

En el caso de los métodos se permitirá incluso redefinir los heredados (polimorfismo con sobreescritura de métodos en subclases).

Variables especializadas

En la diapositiva puede apreciarse que cada clase derivada de la clase Automotor tiene SUS propias variables, además de las de la clase general : placa y velocidad.

Métodos especializados

En el caso de los métodos, algunas de las clases derivadas presentan métodos que sólo tienen sentido en la correspondiente subclase :

levantaEscalera() es un método de la clase CarroBombero.

levantaCarga() es un método de la clase Tractor.

Polimorfismo

Tal como se vio anteriormente, el polimorfismo en OOP puede definirse de dos maneras:

- Sobreescritura de métodos en las subclases
- Sobrecarga de métodos en las clases

Sobreescritura de métodos (Overriding)

Puede sobreescribirse (volver a especificarse) un método en cada subclase (clase derivada) de una clase, de manera que cuando sea invocado, se ejecute de acuerdo a la subclase del objeto en la invocación.

En un árbol de herencia las variables, métodos y constructores son usualmente sobreescritos, esto es que llevan el mismo nombre que los de la superclase. Al momento de hacer referencia a un miembro, generalmente se requerirá especificar el identificador (variable de referencia) de la subclase más reciente, así :

IdentificadorDeObjeto.Miembro

donde Miembro es una variable o método.

Sin embargo en algunas circunstancias será necesario referirse al miembro de la superclase. Asimismo existirán ocasiones donde sea necesario establecer claramente que se quiere hacer referencia al nivel de especialización del momento.

Para estos casos y con el fin de resolver esta situación, donde dos clases presentan iguales identificadores para sus miembros, Java provee las facilidades de referencia this y super.

Selección de métodos en forma dinámica

Tal como se vio, para invocar a un método se utiliza el operador punto con la variable de referencia de un objeto.

Cuando esto se especifica, el tipo (clase) declarado de la referencia de objeto se comprueba durante la compilación, para asegurarse de que el método declarado existe en la clase declarada.

Durante la ejecución, la variable de referencia se podría referir a una instancia de alguna subclase del tipo de referencia declarado. En estos casos Java utiliza la instancia real para decidir qué método llamar en el caso de que la subclase sobreescriba el método al que se invoca.

En la diapositiva puede observarse que :

La clase C es subclase de A.
La subclase C sobreescribe el método m() de A.
En el método main() se ha declarado una variable de referencia "objetoC" del tipo A, pero se ha almacenado en esta una referencia a una instancia de la clase C.

El compilador verificará que A tiene un método llamado m(), pero el intérprete observa que la referencia es realmente una instancia de C, por lo que llama al método m() de C (tal como se aprecia en la salida de la ejecución).

En caso que el tipo con el que se declara una variable de referencia no tiene un método, se irá "barriendo" el árbol de herencia hacia las superclases hasta llamar al método contenido en una clase a la que se extendió.

En el ejemplo de la diapositiva, esto puede apreciarse al invocar al método m() de un objeto de la subclase F, la que no tenía una implementación (sobreescrita) de dicho método. El método que finalmente se ejecuta es el de la superclase A.

this

La palabra reservada **this** se usa para referenciar al objeto del instante de la invocación.

Sintaxis:

this.miembro

super

La palabra reservada **super** se usa para referenciar a la superclase de la clase del momento.

Sintaxis:

super.miembro

Dicho de otra forma :
this hace referencia al objeto del momento como una instancia de la clase del momento.
super hace referencia al objeto del momento pero como una instancia de la superclase de la clase del momento.

Sobrecarga de métodos

Puede llamarse con el mismo identificador, diferentes implementaciones de un método en una misma clase, la diferencia debe encontrarse en los parámetros. El objeto "sabrá" qué implementación ejecutar por el mensaje recibido (por los parámetros).

Como ya se vio, esto también es aplicable al método constructor. En la diapositiva, el método Circulo tiene dos implementaciones: Una sin parámetros, donde siempre crea al círculo con un radio fijo y la otra con parámetro el radio, que permite especificar el radio con el que el círculo se crea.

Paquetes

Paquete es un medio de encapsulamiento de mayor nivel que las clases.
Un paquete puede contener clases y también a otros paquetes.
Los paquetes también agrupan interfaces (luego se estudiará este concepto).
También es un mecanismo para evitar conflictos entre nombres: Puede tenerse dos clases con el mismo nombre pero en diferentes paquetes.
El API de Java es un grupo de paquetes.

Sintaxis:

package IdentificadorDelPaquete[.paq2[...]];

Debe declararse al inicio de un fuente de compilación, antes que cualquier declaración de clase. Estas últimas estarán contenidas en el paquete. Puede observarse que pueden definirse subpaquetes (paquetes dentro de paquetes).

Uso de clases de otros paquetes

Referencia explícita :

Paquete.subpaquete.Clase.Miembro

El inconveniente es que debe colocarse toda la referencia cada vez que desee usarse un miembro o clase.

Importación de paquetes

Permite importar clases de otros paquetes (o incluso paquetes) a una unidad de compilación.

sintaxis:

import IdentificadorDeLaClaseOPaqueteAImportar;

sintaxis para importar todas las clases de un paquete :

import Paquete.subpaquete.*

// no puede usarse * para importar todos los subpaquetes

Visibilidad de miembros desde el exterior de la clase

Desde un principio se ha mencionado que una de las características más importantes de la OOP es el encapsulamiento.
Como la unidad mínima de abstracción es la clase, todas las variables y métodos (miembros) son visibles para todas las demás partes de la misma clase.

Parecería que lo lógico fuese que por lo menos las variables no pudieran ser visibles fuera de la clase. Sin embargo por flexibilidad y porque se tiene un nivel mayor de encapsulamiento (los paquetes), en principio (por comportamiento de los modificadores por omisión) :

Las variables pueden ser accedidas por métodos de otras clases del mismo paquete.
Igualmente, los métodos de una clase son visibles (pueden ser invocados) por métodos de otras clases del mismo paquete.

Entonces, el nivel de encapsulamiento por omisión es el paquete.

Si, según al paradigma de objetos, los datos de estos no deben ser manipulados directamente, sino a través de sus métodos :

Entonces cómo hacer para encapsular variables en una clase ?

Cómo hacer para que sólo algunos métodos sean visibles (para que conformen la interfaz de la clase) y otros no (sean sólo internos a la clase) ?

Modificadores de acceso para miembros

Los modificadores de acceso definen niveles de visibilidad entre miembros (variables y métodos) de una clase y otros objetos.
El modificador de acceso se especifica inmediatamente antes que el tipo de una variable de instancia o antes del tipo de retorno de un método.

Sin modificador de acceso (por omisión)

Si no se especifica modificador, las variables y métodos de una clase sólo podrán ser accedidas por clases del mismo paquete.

public

Especifica que las variables y métodos de una clase podrán ser accedidas por todos los objetos, aunque sean de clases de otros paquetes diferentes al de la clase.

protected

Especifica que las variables y métodos de una clase podrán ser accedidas, además de los métodos de la misma clase, por los métodos de todas las clases del mismo paquete y por las subclases de ésta aunque no se encuentren en el mismo paquete.

private

Especifica que las variables y métodos de una clase podrán ser accedidas sólo por los métodos de la misma clase.

Si una clase tiene modificador de acceso public (como se verá a continuación), puede restringirse el acceso a los miembros de esta clase, anteponiendo los correspondientes modificadores en las declaraciones de variables y métodos. Con esto se sobrepone (o sobreescribe) el modificador de la clase.

Modificador de acceso para clases

Como se vio, la visibilidad de miembros de una clase se determina o modifica en relación a otras clases.

La visibilidad entre clases está determinada en relación a los paquetes.

Por omisión una clase es visible a todas las demás que se encuentren en el mismo paquete.

Si se requiere que una clase sea visible fuera del paquete donde pertenece, debe especificarse el modificador **public** en la declaración de la clase.

Sintaxis :

public class IdentificadorDeLaClase { ... }

En los ejemplos de la diapositiva :

La variable v1 del objeto de clase C1 es visible a la clase C1corrida porque la clase C1 es visible (por encontrarse en el mismo paquete paq1) y porque v1 es pública.

Desde la clase C2 (que está en otro paquete paq2) esta misma variable no hubiera sido visible, por más que es pública, ya que la clase C1 no es visible por encontrarse en otro paquete.

En cambio la variable v1 del objeto de clase C1publica si es visible por ser ésta una clase con acceso modificado a public.

static : Variables de clase

En algunas ocasiones las clases presentan propiedades que son siempre comunes a todas las instancias. Con el modificador **static** se especifica que una variable es común (la misma) para todas las instancias de la clase.

Sintaxis:

```
static tipo identificadorDeVariable;
```

Por omisión (sin modificador), cada objeto (instancia de clase) reserva memoria para sus variables de instancia cuando es creado; por cada nuevo objeto, nuevos espacios de memoria son reservados. Las variables estáticas reservan un sólo espacio independientemente de cuántas instancias de la clase se hayan creado o vayan a crearse y son comunes a todas las instancias de la clase. Es por esto que también se les conoce como **variables de clase**.

Entonces, como la variable es compartida, cambiar su valor implica cambiar la propiedad de todas las instancias existentes de esa clase.

En el ejemplo de la diapositiva se defina una clase CuboEstatico que tiene como variable estática a *lado*. Esto indicará que todos los cubos en un determinado instante tendrán el mismo valor de esa variable.

Como la variable de clase se crea cuando la clase se declara no es necesario que existan instancias para hacer referencia (acceder) a una variable de instancia.

Asimismo, para hacer referencia a éstas, puede usarse el identificador de la clase o también el de un objeto de esa misma clase.

static : Métodos de clase

Los métodos de una clase son generalmente invocados para actuar sobre una instancia determinada de esa clase; esto es, para que modifiquen las variables de instancia de una instancia en particular.

En ocasiones se requiere funcionalidad fuera del contexto de instancias. Con el modificador **static** se especifica que un método es "independiente" de cualquier instancia, por lo que también sólo podrá acceder a variables estáticas y podrá invocar directamente sólo métodos estáticos de su clase.

Al igual que una variable de clase, puede ser accedida sin que exista una instancia, un método estático o método de clase, puede ser invocado sin necesidad que existan instancias.

Asimismo, para invocar a un método estático puede hacerse referencia (calificar el método) a la clase o a algún objeto existente de ésta.

Los métodos static no pueden referirse a *this* ni *super*.

El modificador static podría verse como la forma con que Java permite definir tanto variables como funciones globales.

En este aspecto Java tiene una ventaja sobre el código de biblioteca y variables globales de C/C++; y es que las clases contienen el ámbito de los identificadores y así se evitan colisiones. C permite que se vuelvan a definir funciones de biblioteca del sistema pues todas las funciones ocupan el mismo espacio de nombres.

final : la versión final de variables y métodos

Con el modificador **final** se especifica que una variable tiene un valor constante (su valor no puede ser modificado) o que un método tiene una implementación constante, esto es que no puede ser sobreescrito por subclase alguna.

Al igual que en el caso de variables static, las variables final no ocupan espacio para cada instancia.

Para una variable o método puede decirse que es la "versión final".

Las variables modificadas con final deben ser inicializadas en la declaración y su valor no puede ser modificado. Transgresiones a cualquiera de las dos restricciones anteriores causan errores en compilación.

Una buena convención es usar identificadores con todas sus letras en mayúscula para variables final.

final : la versión final de clases

El modificador **final** también aplica a la declaración de una clase. En este caso especifica que la clase no puede extenderse, esto es que no puede tener subclases.

abstract : implementación de clases abstractas

Una clase abstracta está compuesta de uno o más métodos abstractos, los que son declarados pero no implementados : No tienen cuerpo, pues las implementaciones se encuentran en (son responsabilidad de) las subclases. Puede tener también métodos no abstractos, los que serían comunes para sus subclases.

Tanto a la clase como a los métodos abstractos debe anteponerse el modificador **abstract** en su declaración.

Bastará que una clase contenga un método abstracto para que tenga que ser declarada también abstracta. Generalmente se requieren por conveniencia en el diseño de aplicaciones.

Restricciones:

1.- No puede tener métodos constructores (de creación de objetos).

2.- Los métodos abstractos no pueden especificarse estáticos.

Restricciones (continuación) :

3.- Métodos privados no pueden especificarse como abstract

Otros modificadores

Para métodos :

synchronized

Especifica que sólo se permitirá un camino de ejecución en un método.
En general, Java permite tener más de una ejecución a la vez en un mismo método (fragmento de código). El modificador synchronized forzará a que todos los hilos "esperen su turno".

native

Especifica que la implementación de un método es externa : nativa en C.
El método no tendrá cuerpo (ni siquiera llaves).

Para variables :

transient

Especifica al compilador que una variable no debe ser serializada.

volatile

Especifica que el compilador generará, cargará y almacenará la variable cada vez que sea accedida, en vez de tomar el valor de un registro.
Aunque el objetivo de este modificador es proveer un acceso seguro a una variable por parte de un hilo (thread), es preferible el uso de métodos sincronizados (synchronized).

Necesidad de clases en la compilación

Tal como se vio, para invocar a un método se utiliza el operador punto con la variable de referencia de un objeto.

En caso que el tipo con el que se declara una variable de referencia no tiene un método, se irá "barriendo" el árbol de herencia hacia las superclases hasta llamar al método contenido en una clase a la que se extendió.

En el ejemplo de la diapositiva, esto puede apreciarse al invocar al método m() de un objeto de la subclase F, la que no tenía una implementación (sobreescrita) de dicho método. El método que finalmente se ejecuta es el de la superclase A.

Por este motivo las clases deberán estar presentes durante la compilación.

En este contexto, para que las clases sean reutilizables y accesibles, el diseño obligaría a ir colocando la mayor parte de la funcionalidad siempre "arriba" de la jerarquía de clases, de manera que los mecanismos estén disponibles para una gran variedad de subclases.

Entonces puede apreciarse que sería de mucha utilidad un mecanismo para "desconectar" la definición de un método, o conjunto de métodos, de la jerarquía de herencias

Necesidad de roles y "herencia múltiple"

Lo que diferencia a una clase de otra son sus propiedades y su comportamiento. A su vez, las similitudes de estas características son las que las agrupan en jerarquías de herencia. Sin embargo, ya se ha visto que una clase sólo puede derivar de una y nada más que una clase, de la que hereda propiedades y comportamiento.

En algunas ocasiones, clases que no se encuentran en el mismo árbol de herencia, requieren compartir características similares. En este contexto, puede apreciarse que sería de mucha utilidad un mecanismo que permita "ignorar" la jerarquía de herencia - restringida a una sola superclase - para poder heredar o, mejor dicho, implementar las características comunes requeridas.

Ejemplo.-
Supóngase que se declaran las clases Animal y Máquina, cada una tiene un método tieneVida diferente según la clase:

```
boolean tieneVida() {return true;} // para Animal
boolean tieneVida() {return false;} // para Máquina
```

De estas clases derivan subclases como Gato, Ostra y Hormiga (de Animal); y Automóvil, Televisor, Tostadora (de Máquina).

Si por algún motivo se requiere expandir el diseño para tratar con objetos sonoros y objetos móviles, podrá apreciarse que la clasificación establecida no se adecuaría a este nuevo esquema : si bien una máquina nunca podría ser un animal, ambos - máquina y animal - podrían ser tanto sonoros como móviles.

El problema no puede resolverse colocando, por ejemplo una clase "Sonoro" como superclase de ambas clases pues : qué se haría con clase "Móvil" ? (recuerde que sólo puede tenerse una superclase).

En Java esta necesidad de herencia múltiple se solucionó parcialmente con una construcción (un tipo compuesto) llamada interfaz, que en pocas palabras es un tipo especial de clase que puede ser usada por otras clases como una "superclase adicional".

interface : Declaración de Interfaces

Una interfaz es parecida a una clase abstracta pero con TODOS sus métodos no implementados, sólo declarados.
A diferencia de una clase abstracta (declarada con abstract), que es una clase parcialmente especificada, una interfaz es absolutamente no especificada : TODOS sus métodos son abstractos (sólo declarados y no implementados).

Una interfaz tampoco tiene variables de instancia, aunque puede contener variables estáticas con acceso final (constantes).

Las interfaces proveen los protocolos para acceder a la clase sin necesidad de ver los detalles de implementación.

Sintaxis para declaración de interfaz :

```
interface IdInterfaz {
 TipoDeRetorno IdMetodo1 (ListaDeParametros1);
 TipoDeRetorno IdMetodo2 (ListaDeParametros2);
 ...
 Tipo IdVariableFinal1 = ValorConstante;
 Tipo IdVariableFinal2 = ValorConstante;
 ...
}
```

Entonces para el ejemplo de la diapositiva anterior, tiene más sentido definir (declarar) las interfaces “Móvil” y “Sonora”, ya que una clase puede implementar más de una interfaz, atribuyéndose así muchos comportamientos.

implements : Implementación de Interfaces

Para que una clase pueda implementar una interfaz, sólo requiere implementar cada uno de los métodos MENCIONADOS (pero no implementados) en la interfaz.
Sintaxis :

class IdClase [extends idSuperClase]
 implements IdInterfaz1, IdInterfaz2, ... IdInterfazN {
 cuerpo-de-la-clase
}

Como una clase puede implementar más de una interfaz, a través de este mecanismo puede especificarse una especie de herencia múltiple, pero sólo a nivel de descripciones de métodos y no implementaciones como en otros lenguajes (p.e. C++).

Para el ejemplo viéndose, sólo las máquinas (subclases de Máquina) sonoras necesitarán implementar la interfaz "Sonoro". Sólo las que pueden moverse necesitarán implementar "Móvil". En la diapositiva se muestra la subclase "Automóvil".

Los animales (subclases de Animal) que emitan sonido necesitarán implementar la interfaz "Sonoro" y los que se muevan necesitarán implementar "Móvil". Una implementación para la clase Gato podría ser :

```
public class Gato extends Animal implements Movil, Sonoro {
  String nombre;
  public Gato() {
    nombre = "Gato";
  }
  public String movimiento(){
    return "patas";
  }
  public String sonido() {
    return "MIAU";
  }
}
```

Al ser una interfaz un tipo, pueden declararse variables de referencia y crearse objetos de ese tipo. Siguiendo con el ejemplo, podría definirse una clase "Oidor":

```
class Oidor {
  String sonidoEscuchado;
  void escuchar (Sonoro objetoSonoro) {        // acepta un objeto de alguna
    sonidoEscuchado = objetoSonoro.sonido(); // clase que implemente Sonoro
  }
}
```

Clases internas

Ya se ha visto la importante relación de herencia entre clases, también llamada de derivación, extensión, especialización, etc.
Existe otra importante relación que puede presentarse entre clases : La de "estar contenida", esto es que, para un determinado contexto, una clase puede estar siempre contenida (ser parte interna) de otra.

Una clase interna es la que se define como miembro de otra clase (que la contiene). Al ser una parte de la clase, puede acceder a los demás miembros de la misma.

Ejemplo.-
Si se quisiera implementar una Juego simulado de billar un posible diseño requeriría la clase "Mesa" para la mesa y la clase "Bola" para las bolas.

En la diapositiva puede apreciarse que se ha definido la clase Bola internamente a la clase Mesa. Note que la clase interna puede acceder directamente a la variable de instancia cantidadDeBolas, a la que incluso actualiza en su método constructor.

Para jugar (la clase JuegoDeBillar), primero se crea una mesa.
Luego se inicializa el juego creando cada una de las bolas, ubicándolas en la mesa.
Etc.

El nombre de la clase interna no es visible fuera de la clase "de nivel superior" (o de nivel de paquete), excepto si se le califica completamente (con el operador punto). Esto facilita la estructuración de clases dentro de los paquetes y permite un mejor control de las relaciones entre clases.

Una clase interna puede mencionar directamente (sin calificar) a variables de instancia de su clase de nivel superior.

Esta facilidad de anidar clases permite a clases de nivel superior presentar una organización al estilo de paquete para un grupo lógicamente relacionado de subclases de nivel secundario.

Paquetes y directorios

Los paquetes no sólo sirven para facilitar la organización de las clases, sino también para el manejo de las mismas durante compilación, donde se requerirán muchas clases de locaciones diferentes. En compilación Java encuentra los correspondientes archivos en bytecode y los pone disponibles para el programa que se desea compilar.

Las clases e interfaces son tipos de datos. Estos tipos y los paquetes son identificados por el programa a compilar mediante la sentencia **import**.

Unidades de compilación

Una unidad de compilación es la base de toda aplicación Java.
Una unidad de compilación tiene tres partes que van en secuencia :
- La declaración de un paquete (el paquete al que pertenece la unidad).
- La importación de paquetes (permitiendo usar clases definidas en otros paquetes)
- La declaración de tipos (área en la que se definen las clases e interfaces).

Directorios

Los nombres de paquetes Java son "traducidos" a una ruta del sistema donde residen, teniendo en cuenta el operador (que en este caso parecería actuar de separador) punto.

Biblioteca de clases

La biblioteca de clases de Java está organizada en paquetes, de los cuales, podría decirse que el principal es el paquete java.lang pues es el núcleo del propio lenguaje Java.

En la diapositiva puede observarse la existencia de muchos paquetes que brindan diversas facilidades, que van desde la superclase para construir applets (programas que corren en navegadores de Internet) hasta soporte a networking y bases de datos relacionales.

El paquete java.lang

El paquete java.lang conforma la base de todas las bibliotecas de clases proporcionadas por Java e incluye elementos que definen al propio lenguaje Java.

Contiene las clases Object y Throwable, que son fundamentales para todas las clases (que están en otros paquetes). Object es la superclase para todas estas.

No es necesario importar este paquete ni hacer referencias explícitas (calificar) para acceder a sus elementos.

La clase Object

Puede afirmarse que Object es la clase más importante, por simplemente ser la superclase de donde absolutamente todas las demás clases (provistas y desarrolladas por el usuario) derivan y heredan variables y métodos.

Al igual que no es necesario importar el paquete java.lang, tampoco es necesario extender explícitamente esta superclase (usar la especificación extends) al declarar una clase.

El método **clone()**

El método clone() crea un clon del objeto que invoca al método, creándole un espacio de memoria y asignando a este los valores correspondientes al objeto "clonado".

El método **finalize()**

Cuando un objeto no es referenciado fuera del ámbito, es automáticamente removido. Sin embargo Java provee la facilidad adicional de definir un método finalize, de manera de incorporar lógica asociada a la destrucción del objeto.
La destrucción de objetos no es inmediata sino está supeditada al proceso de recolección de basura de Java, el mismo que es ejecutado a intervalos no controlados por el programador, por lo que incluso el método finalize será ejecutado sólo en ese entonces.

El método **toString()**

El método toString() es proporcionado para ser sobrrescrito y retorna una cadena representando el valor del objeto. Como los valores de los objetos dependen de la clase a la que pertenezcan, cada clase debe sobreescribir este método para brindar información que pueda ser útil por ejemplo a la hora del desarrollo y depuración de código (debugging)

Las clases para "envoltura" de tipos primitivos

Ya se ha visto que Java utiliza tipos primitivos por razones de rendimiento. Estos tipos de datos no forman parte de la jerarquía de objetos y se pasan por valor a los métodos, no siendo factible pasarlos directamente por referencia.

No es posible que dos métodos se refieran a la misma instancia de un int.

Existen algunas clases que - sus métodos - sólo manejan objetos. Entonces para aprovechar la funcionalidad de estas clases por variables de tipo primitivo, se requiere de un mecanismo que permita a estas representarse como objetos.

Java proporciona clases para "envolver" cada uno de los tipos primitivos transformándolos en objetos "referenciables".

Métodos comunes a todas las clases de envoltura excepto a Void:

El método constructor de un método envoltura sólo tiene como parámetro el tipo de dato que está envolviendo.

El método ttttValue(), donde tttt es el tipo envuelto, se usa para obtener el tipo primitivo.

La clase String

La clase String encapsula la estructura de datos y funcionalidad para manejar cadenas de caracteres inmutables, esto es que su valor no puede ser modificado.

Entonces, el contenido de una instancia String no puede cambiarse luego de ser creada, aunque una variable declarada como referencia a String si puede cambiar en cualquier momento para apuntar a un objeto String diferente.

En el ejemplo de la diapositiva, las tres cadenas que se van formando son objetos diferentes, aunque se usa una sola variable "cadena" para referenciar a todas en diferentes momentos.

En la diapositiva también se muestra la gran variedad de métodos que ofrece esta clase. A continuación una breve descripción de algunos de estos.

length() Longitud: cantidad (entera) de caracteres Unicode.
charAt(int *i*) Carácter en posición i (la 1ra posición es la 0).
startsWith(String *prefijo*) boolean: comprobación si cadena comienza con *prefijo*.
startsWith(String *prefijo*, int *posicion*) Idem, a partir de la *posición*.
endsWith(String *sufijo*) boolean: comprobación si cadena termina con *sufijo*.

indexOf(parámetro/s) varias versiones
Posición donde es encontrado el parámetro (sea caracter o cadena).
lastIndexOf(parámetro/s) varias versiones
Posición donde es encontrado el parámetro relativo al final.
substring(int *i*) subcadena desde la posición *i* hasta el final.
substring(int *i*, int *f*) subcadena desde la posición *i* hasta la posición *f*.
equals(Object *o*) boolean: comprobación de igualdad entre cadenas.
equalsIgnoreCase(Object *o*) Idem, sin "sensibilidad" mayúsculas/minúsculas.
compareTo(String *s*) Entero: resultado de la diferencia del primer carácter diferente desde el inicio (izquierda) de la cadena.
concat(String *s*) Cadena que con la cadena *s* concatenada.
replace(char *a*, char *n*) Reemplaza todas las ocurrencias del carácter *a* por *n*.

valueOf(parámetro) Cadena con el parámetro convertido a String.

La clase StringBuffer

La clase StringBuffer encapsula la estructura de datos y funcionalidad para manejar cadenas de caracteres mutables, esto es que su valor puede ser modificado.

En la diapositiva se muestra la variedad de métodos que ofrece esta clase. A continuación una breve descripción de algunos de estos.

length() Longitud: cantidad (entera) de caracteres Unicode.

capacity() Capacidad: cantidad (entera) de caracteres almacenados en memoria. A veces es un número mayor que la longitud pues, a pesar que los caracteres se van asignando a medida que se van necesitando, a veces se asigna más memoria.

setLegth(int *l*)	Cambia la longitud de la cadena invocada.
charAt(int *i*)	Carácter en posición i (la 1ra posición es la 0).
setCharAt(int *i*, char *c*)	Reemplaza el carácter de la posición *i* por el caracter *c*.
append(parámetros)	varias versiones Agrega al final de la cadena invocada el parámetro.
insert(int *p*, parámetro)	varias versiones Inserta el parámetro en la posición p del objeto invocado.

El paquete java.awt (Abstract Windowing Toolkit)

Contiene clases de objetos gráficos útiles para interfaces al usuario (GUIs)

De una forma general, este paquete puede dividirse en tres tipos de clases:
- Componentes
- Layouts
- Utilitarios

Componentes (la clase Component)

Un componente es un elemento gráfico que sirve de interfaz al usuario de una aplicación. También es conocido como un "control"

La mayoría de componentes derivan (son subclases) de la clase Component.

La clase Component es una clase abstracta que define elementos comunes como las propiedades fuente y color y métodos como Mostrar, cambiar de forma y manejar eventos
En la diapositiva puede apreciarse que todas las subclases de Component pertenecen al paquete java.awt a excepción de la subclase Applet, que pertenece al paquete java.applet

Componentes simples

Button (Botón)

Acepta el evento clic para ejecutar una acción asociada

CheckBox (Caja de selección inclusiva)

Representa el estado de una variable lógica (boolean)

Choice (Campo con lista)

Se usa para seleccionar una de muchas opciones especificadas con cadenas de texto. Sólo la seleccionada es mostrada.

Label (etiqueta)

despliega texto

List (Lista)

Se usa para seleccionar una o muchas opciones especificadas con cadenas de texto. Se muestran una cantidad de opciones, de acuerdo a las dimensiones de la lista.

ScrollBar (Barrido del total)

Se usa para representar un valor numérico acotado, que a su vez representa la parte de un componente que es visible en un contenedor.

Componentes de texto (la clase TextComponent)

TextField (Campo de texto)

Componente para edición de texto en una sola línea

TextArea (Área de texto)

Componente para edición de texto que soporta múltiples líneas y barrido vertical

Contenedores (la clase Container)

Un contenedor es un componente que puede contener a otros componentes (incluso a otros contenedores). Todos los contenedores del paquete awt derivan de la clase abstracta Container y los principales son Panel, ScrollPane y Window.

Panel

Elemento usado para agrupar componentes en áreas específicas de la pantalla. Provee la utilidad de posicionar sus componentes en layouts diferentes a los que su contenedor original utilizó. Generalmente el contenedor original es una Window o un Applet

Un **Applet** es una clase de Panel con propiedades adicionales relativas a implementación en browsers.
Uno de los principales métodos de Panel es add(), cuya función es añadir componentes dentro de este contenedor.

ScrollPane

Una forma de panel que permite elementos "hijos" (contenidos) de mayores dimensiones que el propio ScrollPane. En este caso el ScrollPane provee scrollbars verticales y/u horizontales para permitir visualizar diferentes partes del componente hijo.

Window

Representa una ventana sin barra de título ni borde. Estas propiedades son proporcionadas por sus subclases : Frame y Dialog.

La clase Windows

Frame

Provee la funcionalidad de una ventana en el sistema operativo nativo, con la excepción que no es posible acceder al handle o PID de la window (ni de otras windows del ambiente).

Dialog

Representa una ventana mostrada por corto tiempo o que requiere información de transición.

Un Dialog puede declararse modal, por implementación o invocando al método setModal(true). Una caja de diálogo modal es aquella que no permite al usuario acceder al frame que la contiene hasta que ésta haya sido removida.

FileDialog

Es una clase especial de Dialog para requerir del usuario un nombre de archivo o ruta de acceso. Es útil para no tener que estar considerando la complejidad de cada ambiente operativo.

Menús (la clase MenuComponent)

La clase MenuComponent es abstracta y de ella derivan todas las clases para asociar menús a otros componentes, así como para brindar la respectiva funcionalidad.

MenuBar

Encapsula el concepto de un menú asociado a un Frame. Esta asociación puede realizarse invocando el método setMenuBar(objetoMenuBar).

MenuItem

Clase para crear los objetos que serán los items (opciones) de cada menú.
Dos de sus tres constructores permiten especificar la etiqueta que se visualizará.

En la diapositiva puede apreciarse una clase tipo Frame a la que se le asocia un menú.

Primero se crean los 4 objetos tipo MenuItem, una barra de menú y el ítem correspondiente a un tipo Menú ("Archivo").

Puede apreciarse el uso del método *add()* para agregar componentes al Frame.

Programación con eventos

Hace ya buen tiempo que las aplicaciones utilizan interfaces al usuario que son gráficas (GUIs), a través de las cuales el usuario interactúa y comunica sus requerimientos a la aplicación y recibe los resultados de ésta.

En OOP las aplicaciones funcionan mediante invocaciones que objetos se hacen unos a otros. Hasta el momento se ha visto cómo definir las clases para generar los objetos, así como la forma de especificar la funcionalidad (o servicios) que proveerán sus métodos. De la misma forma que estos diseños de la dimensión "estática" del sistema, se hará necesario diseñar la manera como este conjunto de objetos interactuarán reaccionando a acciones del usuario, del sistema operativo o a otros objetos de software del ambiente.

En este contexto se diseña la dimensión "dinámica" de la aplicación, la que, como ya se mencionó, definirá la forma de reaccionar a eventos externos a la misma y además la secuencia y estados por los que pasarán los diferentes objetos al comunicarse (invocarse) unos a otros.

Los objetos han sido diseñados para funcionar cuando son invocados (alguno de sus métodos). Entonces cómo empieza a funcionar una aplicación si todos los objetos están esperando ?
Precisamente cuando sucede alguno de estos eventos mencionados.

La programación basada en eventos tiene la ventaja de representar el permanentemente visible fenómeno de causa/efecto (o estímulo/respuesta) presentado entre los objetos del mundo real. El flujo de control del programa ya no es, como antiguamente, una secuencia definida, sino más bien interacciones entre agentes externos a la aplicación (uno de ellos el usuario) con objetos de software y estos con los propios creados durante la ejecución.

Ejemplos de eventos :
Un usuario produce un evento manipulando un componente gráfico mostrado por la aplicación.
Un dispositivo de entrada (por ejemplo el mouse) genera a su vez un evento para el sistema operativo donde corre la aplicación.
Cualquier otro dispositivo (como por ejemplo alguno para control automático, como podría ser un reloj , termómetro, etc.) también puede producir eventos aceptados por el sistema operativo.

A pesar que en los casos anteriores el S.O. actúa sólo de "intermediario" de los eventos producidos por los agentes externos, éste también puede ser fuente de eventos para la aplicación, cuando efectúa operaciones de entrada/salida, interrupciones de hardware o errores de programa.

El modelo por herencia

Una forma de manejar los eventos en una aplicación es aprovechar la jerarquía de herencia de las clases para tener algunas subclases "especializadas" en manejar eventos. Mientras que el respectivo método (sobreescrito) no indique que puede procesarse el evento detectado, éste objeto evento subirá por la jerarquía de clases hasta lograr que alguna de las superclases indique el evento como "consumado".

Versiones de Java anteriores a la 1.1 utilizaban este modelo para manejar eventos :

Sólo subclases de la clase Component (componente) manejaban eventos sobreescribiendo los métodos action() (para botones, y otros objetos cuyo objetivo es indicar una acción) y handleEvent() (para barras de desplazamiento). El retorno true desde uno de esos métodos consumía al evento, sino se propagaba hacia las superclases de la jerarquía del componente gráfico.

Bajo este esquema los eventos son entregados a componentes, independientemente que estos puedan manejarlos o no. Por este motivo el modelo de herencia no es el más conveniente para manejar eventos ya que se requiere lógica para procesar diferentes tipos de eventos y el programador debe escribir sus propias clases manejadoras de eventos.

El modelo por delegación

La versión 1.1 de Java reemplaza el modelo de herencia por uno de delegación de funciones :

Objetos de cualquier tipo (clase) pueden registrarse como sensibles a eventos (**Listeners**) realizados sobre otros objetos fuente. Las propiedades de notificación de un evento se encapsulan en la clase **Event**. Las notificaciones son propagadas del objeto fuente al Listener mediante una llamada a un método de éste último.

Es el objeto fuente quien recibe directamente el evento de un agente externo y verifica si "es de interés" de uno o más Listeners. Crea un objeto evento (tipo Event) y llama al método apropiado de cada Listener "interesado" para pasarle el evento.

En realidad el objeto evento no hace sino propagar notificaciones de cambio de estado del objeto fuente hacia los Listeners registrados. Cada tipo de notificación de evento es un método diferente. Estos métodos son agrupados (declarados) en interfaces que son extensiones de la interfaz EventListener del paquete java.util. Las clases que "estén interesadas" en un conjunto particular de eventos deben implementar el correspondiente conjunto de interfaces.

Entonces - y en resumen - se requieren tres cosas para recibir eventos :

1.- Declarar una clase Listener, esto es que implemente una interfaz que declare los métodos para recibir cierto tipo de eventos. Crear un objeto Listener (de esa clase).

2.- El objeto debe registrarse con un componente fuente : los componentes fuente ofrecen métodos para esto.

3.- La clase listener debe implementar los métodos de la interfaz, en los que se especifica cómo procesar el evento.

El paquete java.awt.events

Como se vio, un evento se propaga desde el objeto fuente (componente) a uno o más Listeners, que son los objetos que se han registrados como interesados en recibir eventos de cierto tipo sobre ese componente.

Un objeto Listener puede ser otro componente (objeto GUI) o también uno que sea sólo funcional, de manera que se encargue del evento. En este último caso puede escribirse código de manera de separar el componente de su funcionalidad.

El paquete java.awt.events provee dos - conceptualmente diferentes - tipos de eventos :

Eventos de bajo nivel

Son aquellos concernientes con el evento externo específico como por ejemplo el clic del mouse o alguno del ambiente operativo.

Eventos semánticos

Concernientes más bien al significado o intención del evento de bajo nivel. Por ejemplo la acción que se desea realizar cuando se hace clic a un botón o los ajustes que se desea realizar cuando se manipula una barra de desplazamiento (scrollbar).

Un evento de bajo nivel, como el clic de mouse, no ejecuta el mismo rol para diferentes componentes, motivo por el cual se necesitan los eventos semánticos.

En la diapositiva puede apreciarse las clases incorporadas en el paquete java.awt.events.

ActionEvent encapsula la noción de "ejecutar alguna acción".

AdjustmentEvent representa la funcionalidad para "ajustar" un valor numérico.

ItemEvent se usa para indicar que el estado de un Item ha cambiado.

Listeners y Adapters

De acuerdo a este modelo para manejar eventos - por delegación de funciones - todo lo que una clase debe hacer para manejar un evento es implementar una cierta interfaz Listener y registrarse con el objeto fuente (componente).

Existen estas interfaces especializadas en "sentir" eventos - los Listeners. De acuerdo al tipo de evento capaz de sentir también puede agruparse a estas interfaces en semánticas y de bajo nivel.

Listeners para eventos de bajo nivel son diseñados para "sentir" varios tipos de eventos (por ejemplo el WindowListener siente la activación, cierre, desactivación y otros eventos). Si se desea que una clase implemente esa interfaz, deberá implementarse cada uno de los métodos que manejan estos eventos.

Para facilitar estas necesarias implementaciones, el paquete java.awt.events provee un conjunto de clases que "implementan" todos los métodos de las interfaces (Listeners) para eventos de bajo nivel, de manera que sólo sea necesario crear subclases que sobreescriban sólo los métodos deseados. De esta manera se evita tener que proveer implementaciones de todos los métodos por el sólo hecho de requerir implementar una interfaz. Esto puede verse como una manera indirecta de hacer uso de las interfaces (como una especie de clases adaptadoras, de allí el nombre).

Clases para ubicación y ajuste de componentes

El paquete awt provee un conjunto de clases manejadoras de layout (Layout managers o que implementan la interfaz *LayoutManager*) que permiten indicar la manera como los componentes deben ser ubicados y/o ajustados al momento de ser mostrados.

Para establecer un layout a un contenedor (objeto de clase *Container*) se debe invocar el método *setLayout(LayoutManager)*. Cuando ya se encuentra establecido el layout al contenedor, cualquier componente que sea agregado a éste, también será agregado al layout del contenedor.

La clase **FlowLayout**
Coloca los componentes en fila mientras quepan en el contenedor, en este último caso los coloca en la siguiente fila. El orden es el de agregado al contenedor. Este layout es el válido por omisión para la clase *Pane* y por lo tanto para *Applet*s.
Provee métodos para especificar espaciamiento entre componentes y tamaños.

La clase **GridLayout**
Coloca los componentes en filas y columnas según el método constructor elegido y los parámetros especificados.

La clase **BorderLayout**
Coloca y ajusta el tamaño de los componentes de manera que llenen completamente el contenedor. Este layout es el válido por omisión para la clase *Window*, por lo tanto *Frame*s y ventanas de *Dialog*o la usarán si no se especifica otro layout.

La clase **CardLayout**
Coloca los componentes, sin ajustar sus tamaños, al estilo de páginas una detrás de otra (sólo se ve un componente a la vez) y provee los métodos first(), next(), previous() y last() para moverse hacia los componentes.

La clase **GridBagLayout**
Es la más flexible pues presenta más opciones. La base es un grid (aunque no es una extensión de *GridLayout*) pero los componentes no están restringidos a celdas e incluso pueden ocupar más de una. Las características y restricciones para cada componente que se agregará a un contenedor con este layout son especificadas con un objeto de tipo *GridBagConstraints*, que se agrega al layout junto con el componente.

Una aplicación con eventos

A manera de ilustración, se muestra una pequeña aplicación, donde podrá apreciarse el uso de objetos de eventos así como la forma como las clases *Frame* y *Dialog* funcionan.

La aplicación creará un objeto tipo *Frame* conteniendo dos botones, uno para responder al evento (semántico) abrir una ventana de diálogo (tipo Dialog) y otro para cerrar el *Frame* y terminar la aplicación. A su vez, la ventana de diálogo contendrá un botón que acepte el evento (semántico) cerrarse a sí misma.

La clase *FrameDeDialogo* (extensión de *Frame*) estará "interesada" en sentir los clics a los botones para realizar algo, por lo que deberá implementar la interfaz *ActionListener*, asimismo estará "interesada" en interceptar los eventos de ventana (por ejemplo para que el usuario también pueda terminar la aplicación cerrando la ventana) por lo que deberá implementar la interfaz *WindowListener*.

El método constructor:

Cuando un objeto de tipo FrameDeDialogo es creado invoca al superconstructor (constructor de la superclase) que usa como parámetro a una cadena de texto que aparecerá en la barra de título de la ventana.

Luego se indica que los componentes del Container se colocarán en fila invocando el método setLayout pasando un objeto de tipo FlowLayout en el parámetro.

El siguiente método invocado, addWindowListener(this) registra a FrameDeDialogo consigo mismo para propósitos de recibir eventos de ventanas.

Luego se crean los dos botones que estarán contenidos en el Frame, a su vez que se registra al Frame como un ActionListener de manera que intercepte ActionEvents sobre cada botón.

El método addActionListener (ActionListener) está implementado en la clase Button.

Se incluyen ambos botones en el Frame con el método add(componente).

El método add() está implementado en la clase Container (de la que Button deriva).

Es necesario llamar al método show() para mostrar la ventana, ya que por omisión las ventanas son creadas invisibles.

El método MuestraDialogo:

Construye una ventana de diálogo (objeto de tipo Dialog) usando el constructor de Dialog que tiene tres parámetros :
1.- El padre (objeto tipo Frame "dueño del Dialog), que en este caso es *this*
2.- El texto (cadena de caracteres, de tipo String) que aparecerá en la barra de título de la ventana de diálogo
3.- El indicador (variable o constante lógica, de tipo boolean) si será modal. En este caso se está especificando true, para que sea modal.

Luego se indica que los componentes del Dialog (subclase de Container) se colocarán en fila invocando el método setLayout pasando un objeto de tipo FlowLayout en el parámetro.

El siguiente método invocado a ventanaDeDialogo es addWindowListener(this) que registra a FrameDeDialogo (this) para recibir eventos de ventanas desde la ventanaDeDialogo.

Luego se crea un botón que estará contenido en el Dialog y se registra al Frame como un ActionListener de manera que intercepte ActionEvents sobre este botón.

Se incluye el botón en el Dialog con el método add(componente).

Los métodos para manejar los eventos :

Para manejar los eventos deberán implementarse algunos de los métodos declarados en los Listeners implementados por el Frame.

El método actionPerformed

Para clics sobre los botones.
Simplemente verifica sobre qué botón se ha hecho un clic para ejecutar la acción correspondiente.

Se utiliza el método getActionCommand (del objeto de tipo ActionEvent) para obtener la etiqueta del botón.
Otra alternativa era utilizar el método getSource (del objeto de tipo EventObject superclase de ActionEvent) para obtener el objeto fuente, así :

```
public void actionPerformed(ActionEvent e) {
  Object componenteFuente = e.getSource();
  if(componenteFuente.equals(botonAbrirDialogo))
     muestraDialogo();
  else if(componenteFuente.equals(botonCerrarDialogo))
     ventanaDeDialogo.dispose();
  else if(componenteFuente.equals(botonCerrarFrame))
     processEvent(new WindowEvent(this, WindowEvent.WINDOW_CLOSING));
}
```

El método dispose() desaparece una ventana y "dispone" de los recursos utilizados para la misma.

Como puede apreciarse, el hacer clic sobre el botón "Cerrar Frame" no cierra directamente el Frame sino más bien genera otro evento del tipo WindowEvent, el que será aceptado por el Frame pues tiene el rol de (implementa) WindowListener, como si hubiese sido presionado el componente nativo para cierre del Frame.

Cuando suceda este evento se ejecutará el método windowClosing, en donde se hace necesario preguntar por la ventana de origen ya que el Frame acepta cierre tanto del Dialog como de él mismo.

Puede apreciarse que debieron ser implementados todos los demás métodos de la interfaz WindowListener, pero como no se requiere aceptar otros tipos de eventos, - además de los de los botones y los propios de las ventanas - sólo será necesario especificarlos con un cuerpo de método vacío ({ }).

Al no ser un applet, la aplicación requiere de un método main para correrse:

```
public static void main(String[] args) {
  FrameDeDialogo fDD = new FrameDeDialogo();
```

```
}
```

El paquete java.swing

El paquete java.swing viene como uno de los que conforman la librería de clases del Java Foundation Class (JFC) de Sun Microsystems.

Incorpora una rica cantidad de clases y controles (componentes), algunos de los que son extensiones de AWT.

Es un paquete moderno que contiene gran cantidad de controles encontrados en los ambientes gráficos de la actualidad como OS2 Warp, Windows, Solaris, etc.

La clase Applet

Un applet es un programa en Java que corre en un navegador de Internet (Web browser) que sea compatible con Java. Reside en el mismo servidor que el documento HTML y al igual que éste viaja a través de la red y corre como "parte del documento".

Otra forma de verlo es como un objeto de la clase o subclase de Applet.
Esta clase deriva de *Panel* y se encuentra en el paquete *java.applet*.

El paquete java.applet contiene tres interfaces:

AppletContext declara los métodos para trabajar con el documento HTML que contiene al applet.

AppletStub sirve como interfaz entre el applet y el ambiente del browser, enunciando métodos para obtener el URL base del documento, así como acceder a los parámetros del applet o a su estado (activo o inactivo).

AudioClip es una abstracción para tocar un clip de audio una o más veces.

La clase **JApplet** es una extensión de Applet que añade soporte para aceptar comportamiento de pintado a componentes dentro del applet, barras de menús de la clases del paquete *swing* y otra funcionalidad que la hace ligeramente incompatible con Applet en el tratamiento de sus componentes (hijos).

Aplicaciones actuales de las applets :

Gráficos animados
Juegos de Vídeos
Exámenes en línea
Despliegues especiales de texto
Reportes con bases de datos

Applets y Aplicaciones

Alguna de la funcionalidad de Java ha sido restringida a las Applets por cuestiones de seguridad :

- Imposibilidad de acceder a archivos locales (de la computadora donde está corriendo). Si se requiere guardar datos, deberá realizarse en el servidor de donde la página (documento HTML) es "servida".
- Imposibilidad de realizar una conexión de red a una computadora diferente al servidor de donde la página Web fue servida.
- No pueden usar bibliotecas dinámicas o compartidas de otros lenguajes.
- No pueden ejecutar cualquier otro programa local, ni siquiera "plug-in"s del Navegador.

Una aplicación Java, es decir cualquier otro programa que no sea un applet, no tiene tales restricciones, sin embargo facilidades como el de verificación del bytecode siguen siendo implementadas.

A diferencia de una aplicación, un applet no cuenta con el método main. Y es que es el navegador quien proporciona este soporte y es el responsable de correr el applet. Como si el browser fuera el main().

Los métodos de un Applet

init()
Para inicialización : Determinar el estado inicial de actividad del Applet. Es invocado por el browser para informarle (al applet) que ha sido cargado al sistema. Es llamado antes de invocar al método start(). Cualquier applet (subclase de Applet) debe sobreescribir el método para inicializaciones.

start()
Es el método que contiene el cuerpo o "lo que hace" el applet. Es llamado luego de la invocación a init() y también cada vez que sea nuevamente visitado en una página Web. No es necesario invocarlo, nunca. El método debe ser sobrescrito.

stop()
Interrumpe la ejecución del applet pero el sistema no dispone de sus recursos, de manera que puede iniciarse (*start*) nuevamente.
Es invocado cuando la página Web que la contiene ha sido reemplazada por otra.
También siempre debe ser invocado antes de destruir (*destroy*) el applet.
El método debe ser sobrescrito.

destroy()
Destruye (en realidad deja que el sistema disponga de) los recursos usados por el applet, tales como memoria, tiempo del procesador o espacio en disco para swap.

isActive()
Devuelve *true* si el applet está activo, de lo contrario devuelve *false*.

paint() y **repaint**()
Aplican a la ventana del applet automáticamente en determinados momentos, como por ejemplo cuando la ventana debe volver a aparecer luego de haber sido cubierta por otra o cuando la ventana del applet ha sido ajustada en tamaño.
Un applet puede invocar a repaint() para actualizar el despliegue cuando sea necesario.
paint() viene de la superclase *Container* y repaint() viene desde *Component*.

getParameter(cadenaDeCaracteres)
Obtiene el parámetro, SIEMPRE String, pasado desde el documento HTML.

Ejecución de un Applet

Un applet se ejecuta desde un documento HTML.

Se codifica el programa (.java) en un ambiente de desarrollo, como por ejemplo el IDE de VisualAge for Java y se genera el archivo en bytecode (.class).

El archivo ejecutable (en bytecode) debe dejarse en el mismo servidor Web (http) de donde se provee el documento HTML. Generalmente se deja en el mismo directorio o en un subdirectorio de aquel directorio donde se encuentra el documento HTML.

Cuando un cliente de la Web acceda a la dirección (URL) donde se encuentra el documento HTML, este será transmitido por la Red y cargado por el browser.

Al encontrar el "tag" <APPLET ... > , el archivo es buscado en el directorio del mismo servidor, cargado por el browser y automáticamente ejecutado por el mismo.

La especificación en HTML

Para incluir (y ejecutar) un applet en una página Web, debe incluirse en el código del documento HTML el "tag" (etiqueta HTML) <APPLET>, así como los siguientes (sólo CODE, WIDTH, HEIGHT son obligatorios) tags, para finalizar con </APPLET>.

CODEBASE
Para especificar el URL (la dirección en el servidor http) del directorio donde se ubica el applet. Si no se especifica el browser busca en el mismo directorio del documento HTML.

CODE
Nombre del archivo que contiene la subclase (de Applet) compilada (en bytecode).

ALT
Para especificar un texto alternativo si el browser comprende el tag <APPLET> pero no puede ejecutar applets en ese momento.

NAME
Para especificar un nombre para la instancia del applet. Es necesario cuando se requiere que otras applets en el mismo documento HTML se comuniquen entre ellas.

WIDTH y **HEIGHT**
Ancho y alto para el área de visualización del applet.

ALIGN
Similarmente que para alineamiento para imágenes (tag IMG de HTML), los valores posibles son : LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM y ABSBOTTOM.

VSPACE y **HSPACE**
Espacio (en pixeles) a dejar por encima y debajo del applet en el caso de VSPACE y hacia los lados en el caso de HSPACE (similar al caso de IMG).

PARAM NAME
Para especificar los nombres de los parámetros a pasar al applet desde el browser.

VALUE
Para especificar el valor correspondiente a un parámetro.

Facilidades de VAJava para desarrollar Applets

Para escribir el código de un applet puede seguirse la misma secuencia que al escribir cualquier clase, solo que en la declaración de la clase deberá especificar que se extiende la clase Applet.

Visual Age for Java proporciona facilidades para escribir applets :

Habiendo seleccionado un paquete, puede usar el menú *Selected*, luego la opción *Add...* y luego elegir *Applet*.

También puede usar el botón derecho del mouse seleccionando un paquete y elegir las mismas opciones (*Add...* y *Applet*).

VAJava presenta ventanas "SmartGuide" que permiten ir especificando las características deseadas para el applet, como por ejemplo si se desea que el applet responda a eventos, deberá indicarse como qué tipo de Listener (o dicho de otra forma, listener de qué tipo de eventos) se registrará el applet con componentes fuentes de esos eventos.

Implementación de un applet

En la diapositiva se muestra la implementación de un applet llamado "Applet1".

Dicha clase debe extender la clase Applet.

Contiene una variable de instancia de tipo String llamada textoFijo que contendrá un texto a mostrar durante la ejecución del applet.

Asimismo el applet mostrará otro texto pero obtenido como parámetro desde el documento HTML. Para esto se utiliza el método getParameter().

Para mostrar ambos textos se invoca l método drawString del objeto de tipo Graphics que es necesario pasar al método paint().

Para dejar en un directorio del servidor de documentos HTML (en el caso mostrado se está utilizando un directorio local) VAJava provee la facilidad Export, que permite especificar el directorio donde se dejará el ejecutable (.class) e incluso provee de un documento HTML desde donde se ejecutará.

Manejo de Excepciones

A pesar que muchos de los lenguajes, incluso los modernos y los que soportan OOP, brindar facilidades para manejar excepciones, subsiste la costumbre de manejar un sólo código de error como retorno de una función/procedimiento/lógica donde puede producirse más de uno.

El desarrollar aplicaciones sin "huecos" (bugs para los países más al norte) es uno de los principales factores, sino el principal, para que un software o sistema de información se diga de calidad. Sin embargo, para lograr esa ansiada y pocas veces reconocida calidad generalmente es necesario invertir tanto o más tiempo en la lógica de tratamiento de errores que en el objetivo mismo de la aplicación.

El éxito que tenga el desarrollador en contemplar la totalidad de situaciones de excepción y su buen manejo garantizará la eficacia de la aplicación.

Pero...
y qué con la eficiencia durante el desarrollo ?
y con la "mantenibilidad" del código (otro factor importante en la calidad del software) ?

Java provee mecanismos para la notificación y manejo de errores y excepciones que, además de apoyar esta eficiencia en el desarrollo, no se desvían del paradigma de objetos visto a lo largo de todo el lenguaje.

Métodos de nivel "bajo" detectan los errores mientras los de "alto" nivel (quienes invocan a los anteriores) deberán manejar estos errores y excepciones, que fueron lanzadas como objetos.

Una excepción es una condición anormal que surge en una secuencia de código durante ejecución.

En Java, una excepción es un objeto que describe una condición excepcional producida en un fragmento de código.

La clase Throwable

La clase *Throwable* representa todas las condiciones excepcionales.

La subclase *Exception* se utiliza para condiciones excepcionales que los programas de usuario deberían capturar. Extendiendo esta subclase se pueden crear condiciones excepcionales propias para que otros las capturen.

La subclase *Error* define las condiciones que no se deberían capturar en condiciones normales.

Sólo instancias de la clase *Throwable* o de sus subclases podrán ser originadas por la máquina virtual de Java o por la sentencia *throw*.

En términos prácticos, la clase sirve para contener una "instantánea" (snapshot) del contenido del stack de ejecución del hilo en el momento que la instancia respectiva fue creada. También puede contener una cadena de texto para ampliar la información del error.

La clase Error

Clase	Descripción
LinkageError	Error en dependencia de clases
ClassCircularityError	No se usa
ClassFormatError	Formato no válida para una clase binaria
ExceptionInInicializerError	Excepción inesperada en inicializador
IncompatibleClassChangeError	Uso inapropiado de una clase
AbstractMethodError	Se trató de invocar un método abstracto
IllegalAccessError	Se trató de acceder a un objeto inaccesible
InstantiationError	Se trató de crear un objeto de una clase abstracta
NoSuchFieldError	No pudo encontrarse el campo especificado
NoSuchMethodError	No pudo encontrarse el método requerido
NoClassDefFoundError	No pudo encontrarse la definición de la clase
UnsatisfiedError	Enlaces (links) sin resolver en la clase cargada
VerifyError	Imposibilidad de verificar el bytecode
ThreadDeath	Indica que el hilo terminará
VirtualMachineError	Superclase para errores de la máquina virtual
InternalError	Error durante la ejecución del intérprete Java
OutOfMemoryError	Faltó memoria
StackOverflowError	Se sobrepasó capacidad del stack
UnknownError	Error desconocido durante uso de máquina virtual

La clase Exception

Clase	Descripción
ClassNotfoundException	No puede cargarse la clase especificada
CloneNotSupportedException	El objeto no implementa interfaz Clonable
IllegalAccessException	La clase no es accesible
InstantiationException	Se trató de crear instancia de clase abstracta
InterruptedException	Hilo interrumpido
NoSuchFieldException	Se trató de acceder campo inválido
NoSuchMethodException	No pudo resolverse el método
RunTimeException	Superclase
ArithmeticException	Condición de error aritmético (p.e. / cero)
ArrayStoreException	Tipo de objeto no concuerda con el del arreglo
ClassCastException	Conversión de tipo inapropiado para el objeto
IllegalArgumentException	El método recibió un argumento inválido
IllegalThreadStateException	Operación inválida para el estado del hilo
NumberFormatException	Imposibilidad de convertir String a número
IllegalMonitorStateException	Inadecuado estado del monitor (sincronización)
IllegalStateException	El método se invocó en un momento inválido
IndexOutOfBoundException	Índice fuera de los límites
ArrayIndexOutOfBoundsException	Índice negativo o mayor al tamaño del arreglo
StringIndexOutOfBoundsException	Índice negativo o mayor al tamaño del String
NegativeArraySizeException	Tamaño del arreglo menor que cero
NullPointerException	Se trató de acceder miembro de un objeto null
SecurityException	Operación no permitida (según configuración)

Excepciones en operaciones de Entrada/Salida

Tanto las aplicaciones que tienen que interactuar con el usuario como aquellas que no, tienen que manejar (recibir y entregar) datos de alguna manera. Las primeras lo hacen desde y hacia dispositivos manipulados por el usuario como un teclado y un monitor, mientras las segundas lo hacen con dispositivos de almacenamiento y comunicación de datos de datos como un disco magnético o un conector (socket) de red.

Java llama "flujo" (stream) a la abstracción correspondiente a la entrada/salida de datos desde y hacia dispositivos, de manera que se encapsula la complejidad de diferenciar los diferentes tipos de dispositivos (el código de programa no debe preocuparse de diferenciar entre un teclado y un socket de red).

La entrada (desde cualquier dispositivo) está encapsulada en la clase ***InputStream*** y la salida en la clase ***OutputStream***. Java también provee subclases de estas precisamente para manejar las diferencias entre dispositivos (archivos de disco, socket y hasta buffers de memoria).

Las operaciones de E/S son tan comunes, que el correcto manejo de las excepciones que puedan originarse es un aspecto fundamental para el éxito en el desarrollo de aplicaciones de calidad. Java provee un paquete completo, ***java.io***, para este tratamiento.

Clase	Descripción
IOException	Superclase para las excepciones de E/S
EOFException	Fin de archivo
FileNotFoundException	No se pudo ubicar el archivo
InterruptedIOException	Operación de E/S quedó interrumpida
SyncFailedException	Falla en la sincronización
UTFDataFormatException	Cadena en formato UTF-8 incorrecta
CharConversionException	Superclase para excepciones de conversión char
UnsupportedEncodingException	Falla en mecanismo para codificar caracteres
ObjectStreamException	Superclase para excepciones de flujos de objetos
InvalidClassException	Clase inválida para serialización
InvalidObjectException	Clase prohibe serialización
NotActiveException	Serialización inactiva
NotSerializableException	Clase no puede serializarse
OptionalDataException	Hay datos adicionales, fin de archivo.
StreamCorruptedException	Fallo en el chequeo interno del flujo
WriteAbortedException	Excepción en el flujo

Un programa con error

Visual Age for Java provee la herramienta "Debugger" para rastreo de errores y excepciones originados en tiempo de ejecución.

El Debugger despliega no sólo el lugar donde ocurrió la excepción sino todo el "stack" o pila de llamadas, que en Java son los métodos y su correspondiente objeto/clase que lo ejecutó.

En el ejemplo de la diapositiva se muestra un pequeño programa que invoca a un método que efectúa una división entre cero, que es una operación matemática inválida.

Puede apreciarse que el Debugger despliega a la clase que se percata de la excepción, que en este caso el mismo sistema Java contempla esta excepción :

La clase *ArithmeticException* lanza esta excepción.

Puede apreciarse también la cadena de llamadas y los lugares precisos donde se originan las excepciones.

Esta es una gran facilidad que ofrece Visual Age for Java al momento de desarrollar y probar aplicaciones.

try y catch : capturando y manejando excepciones

Si bien el sistema (el intérprete Java) es muy eficiente y - tal como se vio - puede capturar diversas excepciones y mostrar el detalle, por lo general lo que se requiere es seguir manteniendo el control del flujo de un programa.

Entonces deberá capturarse y manejarse la excepción.

try se utiliza para especificar el bloque que debe protegerse frente a todas las excepciones.
inmediatamente al bloque *try* se incluye la cláusula ***catch*** para especificar el tipo de excepción que se desea capturar.

El ámbito de la cláusula *catch* está restringido a las sentencias del bloque *try* precedente.

Lo lógico es construir bloques *catch* que resuelvan la condición excepcional, por ejemplo dejando a las variables en un estado de manera que pueda continuarse con la ejecución. Si un método no maneja una excepción, entonces será decisión de quien lo invocó el manejarla. Así se "sube" sucesivamente por la cadena de llamadas hasta que se llegue al manejador (en tiempo de ejecución) de Java, quien detalla todas las llamadas.

En ocasiones, un mismo bloque puede originar más de una condición excepcional, entonces puede colocarse varias cláusulas *catch*, las que se comprobarán en la secuencia que fueron especificadas, por lo tanto es recomendable que las clases de excepción más específicas sean colocadas primero.

En la diapositiva puede apreciarse que sin parámetros (la primera ejecución o "corrida") el divisor se hace cero y se origina una excepción de división por cero.

En la segunda corrida se ha proporcionado un parámetro entonces el bloque logra el "intento de división" pero arroja otra excepción ; el índice del arreglo está fuera del rango.

try anidadas

Los bloques try se anidan de forma similar a los ámbitos de las variables. En un bloque try puede especificarse una llamada a un método y en este último puede colocarse la sentencia try para proteger otro bloque. Así sucesivamente.

Si un método no tiene un manejador (cláusula catch) para un tipo de excepción, se sube por la cadena de llamadas (al bloque try que invocó), hasta llegar incluso al manejador de excepciones de Java.

throw : lanzando excepciones

La sentencia ***throw*** se utiliza para lanzar explícitamente una excepción.

Para lanzarla, debe tenerse la excepción (una variable de referencia a una instancia de la clase *Throwable*) a lanzar. Esta puede ser la capturada en un *catch* o también una nueva (con el operador *new*).

Sintaxis :

```
throw objetoThrowable;
```

El flujo de programa no continua con la siguiente instrucción sino que se inspecciona el bloque try-catch de su nivel para ver si la excepción del catch coincide con el tipo de la instancia Throwable del throw. Si no, se inspecciona el bloque try-catch que la engloba y así hasta llegar incluso al manejador de excepciones de Java.

La sentencia *throw* siempre termina el método y le brinda a quien lo invocó la oportunidad de capturar la excepción.

En el ejemplo de la diapositiva se crea una (nueva) excepción, que luego de ser manejada por el bloque catch, es lanzada al bloque externo que la llamó.

throws : excepciones no capturadas

La sentencia ***throws*** se utiliza para especificar la lista de excepciones que un método puede lanzar.

Sintaxis :

```
tipo metodo (argumentos) throws listaDeExcepciones { ...
```

Para la mayoría de las subclases de Exception Java obligará a especificar esta lista de excepciones o sino a manejarla (try-catch) en el método.

Esta regla no se cumple para la clase (y subclases de) *Error* ni *RunTimeException*.

Si en el ejemplo de la diapositiva mostrada, no se hubiese especificado *throws* en la especificación del método *metodo()* , Java ni siquiera permitiría pasar compilación.

finally : independizando código de las capturas

La cláusula ***finally*** se utiliza para asegurar que un bloque de código se ejecutará, independientemente de las excepciones que se produzcan y/o que se capturen.

Este bloque se ejecutará aunque ninguna de las cláusulas *catch* haya logrado atrapar una excepción y antes que se ejecute la siguiente sentencia al bloque try-catch.

Siempre que un método vaya a devolver el control a quien lo invocó, mediante una excepción no capturada o con un retorno explícito (con la sentencia *return*), se ejecutará el bloque finally antes del final del método.

Aunque la cláusula *finally* es opcional, será muy útil en caso de haber asignado muchos recursos en un método donde pueden ocurrir excepciones; el bloque *finally* podrá servir para liberar los mismos.

Hilos y sincronización

Java soporta multithreading desde su propio ambiente. Este concepto puede definirse como la capacidad de un programa para ejecutar varias acciones en forma concurrente.

Los hilos son flujos de ejecución dentro de un mismo programa o applet.

Los hilos en una aplicación comparten recursos del sistema, a diferencia de cuando se tiene múltiples aplicaciones que generalmente tienen su propia copia de datos y programa.

Dentro de la aplicación, cada hilo es controlado individualmente y tienen un inicio y final definidos. Generalmente son "pre-emptivos", esto es que pueden interrumpirse, empezar o detenerse en cualquier momento.

Los hilos interactúan con el ambiente "run-time" de Java respondiendo a los eventos que ocurren. Cuando se crea una clase derivada de la clase **Thread** o que implementa la interfaz **Runnable**, se especifican métodos que determinan cómo el programa responde a estos eventos.

Java siempre crea por lo menos un hilo de ejecución. Efectivamente, para aplicaciones con el método main, la máquina virtual de Java crea un hilo.

Las partes de un thread

El "cuerpo" del hilo

Es el método run(), que puede establecerse de dos maneras:

1.- Creando una subclase de Thread y sobreescribiendo run()

2.- Creando un hilo con un objeto destino que implemente la interfaz Runnable (y que deberá implementar run()).

En el ejemplo de la diapositiva se usa al mismo objeto (this) pero podría ser cualquier objeto Runnable (que implemente esta interfaz).

A diferencia de main() que es static (pues sólo hay una ejecución de main), run() no debe ser static, su ejecución estará siempre asociada a un objeto : un objeto de la clase (o subclase de) *Thread*.

Los estados de un hilo

Un hilo puede encontrarse en diversos estados, que especifican la condición de uso del hilo y su capacidad para correr.

NUEVO (New)
Cuando se crea un objeto tipo Thread se encuentra vacío, esto es que no se le han asignado recursos del sistema.
En este estado sólo puede iniciarse (start) o detenerse (stop).

```
Thread unHilo = null;
```

EJECUTABLE (Runnable)

Este es el estado cuando el hilo se encuentra ejecutando las sentencias del método run(). NO SIGNIFICA QUE EL HILO ESTE CORRIENDO pues podría encontrarse esperando la ejecución de un hilo de mayor prioridad. El estado se logra luego de la invocación del método start(). Realmente el hilo se encuentra "elegible" para ser ejecutado.

NO EJECUTABLE

Cuando se encuentra esperando que ocurra algún evento.
Para cada condición de NO EJECUTABLE sólo existe una forma de salir o cambiar de estado a EJECUTABLE :

Condición	Evento que cambia al hilo a EJECUTABLE
Dormido (sleeping)	Tiempo especificado expira.
Suspendido (Suspended)	El mensaje (método) resume() es recibido.
Condicionado (Condition)	El mensaje notify() o notifyall() es recibido.
En espera de I/O	Una acción de Entrada o Salida es completado.

MUERTO (Dead)

Un hilo "muere" cuando recibe el mensaje (método) stop o cuando concluye satisfactoriamente la ejecución de su cuerpo (run), por ejemplo al haber concluido un bucle.
Se puede utilizar el método boolean isAlive() para conocer el estado de un hilo.

Programas de un sólo hilo

Para ser ejecutado en un browser, la clase Reloj tiene que extender (derivar de) la clase Applet.
De manera de actualizar y mostrar la información permanentemente, se incorpora un hilo al applet.

El método start es invocado cuando el browser carga el applet.
Si el hilo es null, el reloj es nuevo o ha sido previamente detenido (stop)
En ambos casos se debe crear un nuevo hilo. En este caso se usa el constructor que requiere dos parámetros : el objeto que va a implementar el cuerpo del hilo (que en este caso es el mismo applet) y un nombre para el hilo ("reloj"). Además se invoca el método start() sobre el hilo.

El método run es el que hace funcionar el reloj. Invoca a repaint() y luego "duerme" 1 segundo (1000 milisegundos). Esto lo hace con un bucle infinito hasta que se detenga el proceso haciendo al hilo = null.

Puede apreciarse que el método *paint()* utiliza un objeto tipo Date para obtener los datos del tiempo.

El método *stop()* es llamado cuando el applet recibe un evento stop, por ejemplo cuando se deja la página Web donde está corriendo el applet (también puede ser invocado ante cualquier otro evento como hacer clic sobre un botón).

En general es recomendable incluir en el método *stop* de toda applet, las instrucciones necesarias para dejar en espera o detener los hilos si es que no es necesario seguir ejecutándolos mientras no se vea la applet.
En este caso particular se dispone del hilo (haciéndolo null) para que el proceso de recolección de basura automático disponga de los recursos.

Programa de varios hilos (Multi-threading)

El manejar más de un hilo (multi-threading) provee el beneficio real de esta facilidad de Java.

En el programa mostrado en la diapositiva se crean dos hilos un hilo para la banda negra y otro para la banda roja.

La constante SALTO se establece en 2 para que se crezca en esa cantidad cada décimo de segundo (100 milisegundos).

La clase contiene una clase interna (*Banda*) extensión de *Thread* y que además aceptará eventos (implementa la interfaz *ActionListener*).

Dos hilos son declarados como objetos de la clase interna (*bandaRoja* y *bandaNegra*).

Cuando una banda es creada, crea y añade un botón al applet y además se registra a sí misma como Action Listener en el botón. Luego el hilo se inicia (start). Los parámetros del constructor de la banda son el texto que irá en el botón asociado y la altura donde se dibujará la banda.

Cuando un botón es presionado el método *ActionPerformed* de la banda correspondiente al botón será invocado.

Este método suspende el thread si la banda está creciendo (si variable=true) o lo reinicia (invocando a resume()) si está suspendido (si variable creciendo = false).

El método *run()* es un bucle infinito que duerme al hilo durante 1/10 de segundo (100 milisegundos) luego incrementa la longitud de la banda en una cantidad igual a la constante SALTO y luego vuelve a mostrar la banda. El bloque try/catch es requerido debido a que el método sleep de la clase Thread arroja una excepción:

```
public static native void sleep(long milisegs) throws InterruptedException;
```

El método *paint*() del applet invoca al método *draw* de cada banda, pasándole un objeto gráfico y el color de cada una.

Cuando el usuario del applet deja la página Web, los dos threads son suspendidos y la longitud de las bandas es dejada en cero.

Cuando el applet es destruida (destroy) deben detenerse los hilos.

Las bandas han sido registradas como procesadoras de eventos, esperando (listening) clics en los botones. La clase exterior no tiene porque incorporar código para manejar los eventos en los hilos.

Este uso del modelo de eventos por delegación y el uso de clases internas permite escribir un código más modular y fácil de mantener. Se evita tener varios bloques condicionales (if, else, switch) para controlar la manera como los eventos son manejados

La interfaz Runnable

La interfaz *Runnable* debe ser implementada por cualquier clase cuyas instancias requieran ser ejecutadas por un hilo (thread).
Fue diseñada para proveer un protocolo común para objetos que requieran ejecutar código mientras estén "activos".

La clase *Thread* implementa esta interfaz.

Adicionalmente Runnable provee la facilidad a una clase para estar activa o "correr" (run) sin necesidad de derivar de *Thread* :
Como ya se vio, la clase debe crear un objeto de tipo *Thread* y pasarse a sí misma como el destino de este hilo.

Generalmente esta interfaz se usará en el caso que sólo se requiera sobreescribir el método run() y no otros métodos de la clase Thread.

Los métodos de la clase Thread

Los constructores :
Thread();
Thread(Runnable destino);
Thread(Runnable destino, String nombreDelHilo);
Thread(String nombreDelHilo);
Thread(ThreadGroup grupo, Runnable destino);
Thread(ThreadGroup grupo, Runnable destino, String nombreDelHilo);
Thread(ThreadGroup grupo, String nombreDelHilo);

parámetro **nombre** :
Todo hilo tienen un nombre, que puede ser especificado usando el constructor que lo requiere como parámetro o es colocado automáticamente en caso de no haberse especificado : Thread-n, donde n es un entero.

parámetro **destino**
Es el objeto Runnable cuyo método run() se ejecutará como el cuerpo del hilo.

parámetro **grupo**
Es el grupo (objeto de clase ThreadGroup) donde el hilo será colocado.

setName() permite modificar este nombre a un Thread.

getName() permite obtener el nombre de un hilo, facilidad para búsqueda de errores durante el desarrollo de aplicaciones (debugging).

start()
método de un hilo que debe invocarse para que se ponga en estado ejecutable.
Si es invocado más de una vez sobre el mismo hilo, una excepción es lanzada.

stop()
causa que un hilo termine, lanzando una excepción hacia éste (una excepción de tipo ThreadDeath). El mismo efecto sería ejecutar :
throw new ThreadDeath()
que aplica al hilo del momento, en cambio stop() puede invocarse desde otros hilos.

Ejecución de hilos y priorización

En Java es la prioridad de los hilos la que establece el orden en que son ejecutados en secuencia.
La CPU sólo puede ejecutar instrucciones de un único hilo a la vez y la prioridad es el mecanismo para cambiar a la ejecución de otro hilo :

Un hilo puede ceder voluntariamente el control (explícitamente, al dormirse o al bloquearse esperando por una E/S pendiente) o puede también ser desalojado.

Si un hilo cede el control, el hilo EJECUTABLE de mayor prioridad se selecciona y asigna a la CPU.

Un hilo es desalojado (independientemente de lo que estuviese haciendo) si otro de mayor prioridad cambia al estado EJECUTABLE.

La prioridad de un hilo (cuando se tienen varios)

La prioridad de los hilos establece el orden en que son ejecutados en secuencia.

Cuando un hilo es creado, hereda la prioridad del thread que lo construyó.

La prioridad puede modificarse enviando el mensaje (con el método) **setPriority().**

Los valores para prioridades van del 1 al 10 y los usados con mayor frecuencia son las siguientes constantes :

```
MIN_PRIORITY   =  1
NORM_PRIORITY  =  5
MAX_PRORITY    = 10
```

El sistema (run time) siempre elige al hilo EJECUTABLE de mayor prioridad para ejecutar.

Si un hilo de alta prioridad deja el estado de EJECUTABLE a NO EJECUTABLE, uno de menor prioridad se ejecutará.
Hilos de igual prioridad siguen turnos para ejecutarse.

Sincronización

En ambientes donde se tienen procesos concurrentes y ya que los hilos permiten y potencian el comportamiento concurrente y asíncrono de procesos, debe existir alguna manera de forzar el sincronismo cuando sea requerido.

La mayoría de implementaciones hacen uso del concepto llamado "Monitor", que, tal como se puede apreciar en la diapositiva, puede entenderse como una caja donde sólo cabe un hilo. Un cerrojo de exclusividad (mutex por sus siglas en inglés) que sólo es de propiedad de un hilo en un instante dado. Cuando un hilo adquiere un cerrojo se dice que ha "entrado" en el monitor y los demás que intentan acceder al mismo quedan "esperando".

Java, en vez de implementar una clase Monitor (como otros lenguajes OO), incorpora el concepto de manera implícita en todos sus objetos.

Todo objeto tiene un monitor implícito en el que puede entrar simplemente invocando a cualquier método sincronizado (precedido del modificador ***synchronized***). Una vez que el hilo está en el método, ningún otro hilo puede invocar métodos sincronizados sobre el mismo objeto.

Para salir del monitor y permitir el control al siguiente hilo en espera, el propietario del monitor sólo tiene que volver (return) del método.

Una aplicación que necesita sincronización

La clase *Invocado* tiene un sólo método llamado *metodo*, que recibe una cadena (*mensaje*) y la imprime entre corchetes. Pero luego de imprimir el primer corchete y el mensaje, detiene el hilo durante un segundo (lo pone a dormir con el método - estático - sleep()).

El constructor de la clase *Llamador* toma como parámetros a una referencia a un objeto tipo Invocado y una cadena. También crea un nuevo hilo que llamará al método de este objeto : *run()*. Este hilo se ejecuta de inmediato pues se pone EJECUTABLE con el método *start()*.

La clase Ejecutora hace funcionar todo :

crea una única instancia de la clase *Invocado* y tres objetos de tipo *Llamador* cada uno con una cadena (para el mensaje) diferente.
La misma instancia de Invocado se pasa a cada Llamador.

El método *sleep()* permitió que los hilos conmutasen de contexto y se mezclaron las salidas de los tres mensajes.

No hay nada que impida que los tres hilos llamen al mismo método, sobre el mismo objeto y al mismo tiempo.

synchronized : el modificador

El modificador para métodos *synchronized* forzará a que todos los hilos "esperen su turno".

En el ejemplo de la diapositiva, una vez colocado este modificador, cualquier hilo que desee ejecutar este método, deberá esperar que el hilo retorne de este método.

synchronized : la sentencia

En algunos casos puede requerirse utilizar clases que no han sido diseñadas para accesos multihilo, por lo que no tienen métodos sincronizados.

En estos casos puede "envolverse" la llamada a uno de estos métodos no sincronizados en un bloque sincronizado especificado por la sentencia *synchronized*.

Sintaxis :

```
synchronized (expresionQueResulteEnUnObjeto) {
        bloqueOsentencia
}
```

Ejecutar el bloque o la sentencia especificada tiene el mismo efecto que invocar a un método sincronizado : el monitor del objeto que resulte de la expresión es "adquirido" antes que un bloque de código pueda ser ejecutado. Este bloque de código generalmente contiene una invocación a algún método del objeto.

En el ejemplo de la diapositiva se hace una variante para obtener el mismo resultado :

Se deja al método *metodo* sin sincronizar utilizando en su lugar la sentencia *synchronized* dentro del método *run()* de *Llamador.*

Se obtendrá el mismo resultado sincronizado ya que cada hilo espera a que el anterior termine antes de empezar.

Componentes de Software

El uso de objetos (en realidad clases) que puedan ser reutilizados dentro de una aplicación puede generalizarse a objetos disponibles por otras aplicaciones que puedan hacer uso de su funcionalidad o, dicho de otra forma, de los servicios ofrecidos. Estos objetos que emulan piezas de un ensamble de procesos industriales son conocidos como **componentes de software**.

Como el objetivo principal es la reutilización de los mismos, los componentes no pueden ser como las clases escritas en las aplicaciones que tienen una interfaz muy específica a la aplicación. Más bien tienen que tener interfaces genéricas y abiertas. Precisamente, para promover la reutilización del software, han sido definidas convenciones para interfaces de clases, también llamadas modelos de componentes.

JavaBeans es el modelo de componentes estándar del lenguaje Java. Asimismo es el modelo utilizado por VisualAge. El modelo incluye :
- Un modelo de eventos para especificar cómo un componente envía mensajes a otro sin necesidad de conocer exactamente los métodos que el objeto receptor implementa. Esto permite que un componente pueda ser utilizado con una diversidad de objetos que implementan diferentes interfaces (conjuntos de métodos)
- Eventos, propiedades y métodos
- Introspección : Capacidad de descubrir en tiempo de ejecución la interfaz de una instancia de una clase de componente.

Beans de Java

Un bean es un objeto (de Java) que cumple la especificación JavaBeans.

Esto es que, como todo objeto Java, presenta propiedades (datos) y métodos (funcionalidad), sin embargo presenta una diferencia muy importante :

Su interfaz (o protocolo de comunicación con el exterior para aceptar mensajes) no se limita a los métodos visibles cuyos parámetros deben ser conocidos y especificados al momento de escribir el programa como en las clases de Java, sino que involucra :

Los eventos que puede originar (y que otros pueden "escuchar", para lo que tienen que registrarse).

El conjunto de valores de la propiedades que pueden ser accedidos (consultados o almacenados).

Los métodos públicos que, agrupados, conforman los "servicios" visibles o disponibles ofrecidos por el bean.

Características y Tipos de beans

La especificación JavaBeans encamina ciertas características para los componentes (beans) bajo su especificación :

1.- Los beans son simples y compactos.
La simplicidad debe darse en los aspectos de creación y utilización de los beans. Los beans deben ser compactos de manera de poder ser utilizados en ambientes distribuidos, donde deberán ser transmitidos en su totalidad a través de conexiones Internet de diversos anchos de banda.

2.- Los beans son portables
Por lo menos a nivel de las plataformas que soportan Java (y en estos tiempos casi todas están comprometidas con Java).

3.- Los beans aprovechan de las características de Java
Por ejemplo el poder ir integrando objetos a una aplicación, independientemente del origen de los mismos o de su historia (cómo fueron desarrollados).
Otra de las características aprovechadas es la *Persistencia* (la capacidad que tiene un objeto de almacenar y posteriormente recuperar su completo estado interno), que es manejada automáticamente en los beans en base al mecanismo de *Serialización* presente en Java.

4.- Los beans brindan soporte para el proceso de desarrollo de aplicaciones
La arquitectura JavaBeans incluye soporte para especificación de propiedades en tiempo de diseño, motivo por el cual las herramientas de terceros aprovechan fácilmente esta característica para integrarlo a sus facilidades visuales.

5.- Los beans no están restringidos a un único modelo de computación distribuida

En este contexto existen - y pueden desarrollarse - beans simples, que ofrecen pocos servicios y presentan una interfaz muy concreta u otros más complejos, que pueden ser contenedores de otros y presentan una variedad de servicios documentada a través de su interfaz.
Asimismo pueden existir bean visuales que poseen propiedades asociadas a una GUI y otros cuya interfaz es sólo funcional.

Integración con Herramientas Visuales

Como ya se mencionó, un bean de Java cumple la especificación JavaBeans, entonces presenta la característica de Introspección.

Esta característica es la que facilita la integración de componentes a herramientas visuales para desarrollo aplicaciones.

Quien desarrolla encuentra beans en la paleta de herramientas y lo que generalmente plantea primero es el aspecto visual de la aplicación (la interfaz al usuario) distribuyendo los beans en esta GUI.

Luego "customiza" los beans modificando valores de sus propiedades. Para esto hace uso del correspondiente "editor de propiedades" provisto por el los propios beans (y aprovechado por la herramienta visual).

Se codifican los bloques apropiados para manejar eventos, esto es las conexiones entre beans y con la aplicación.

java.beans : el paquete del API de JavaBeans

En la práctica, JavaBeans es una interfaz programática y sus facilidades han sido implementadas a través de las clases agrupadas en el paquete java.beans.

Estas facilidades o servicios pueden agruparse de la siguiente manera :

1.- Manejo de propiedades
2.- Introspección
3.- Manejo de eventos
4.- Persistencia
5.- Soporte para construcción de aplicaciones

Manejo de propiedades

Las propiedades reflejan el estado de un bean y constituyen la parte de DATOS de su estructura. Determinan su apariencia (obviamente en el caso de beans visuales) y también su comportamiento, ya que éste puede variar de acuerdo al estado en que se encuentre.
Las propiedades pueden ser accedidas y usadas :
- A través de lenguajes interpretados como JavaScript o VisualBasic Script y obviamente a través de lenguajes completos para OOP como Java.

- "Programáticamente", a través de sus métodos públicos especiales para acceder propiedades. La mayoría de beans presentan métodos para recuperar lo valores de sus propiedades (y que generalmente empiezan su denominación con **get**... por su significado en inglés) o para fijarlos en algún valor (empiezan con **set**...).

- Visualmente, a través de su integración con herramientas visuales que presentan editores de propiedades.

- A través de las capacidades de persistencia, donde puede almacenarse y recuperarse los valores de las propiedades del bean.

El API JavaBeans soporta propiedades para índices, esto es tratamiento de arreglos de manera similar a los de Java (índices enteros). Útiles cuando se requiere grupos de propiedades del mismo tipo. Por ejemplo un bean contenedor de otros beans puede almacenar la distribución de estos en un arreglo.

El API JavaBeans también soporta propiedades que pueden generar notificaciones cuando cambian de valor. La notificación llega a un applet, aplicación u otro bean que esté interesado en este evento.

Introspección

El API de JavaBeans provee mecanismos que permiten conocer la estructura interna de un bean. Para esto utiliza servicios de reflexión (reflejo de características internas) de bajo nivel y facilidades que han sido implementadas a través de la clase ***Class*** (del paquete *java.lang*) y las clases agrupadas en el paquete *java.lang.reflect.*

En realidad, al construir nuevos beans, se estará utilizando indirectamente el paquete *java.lang.reflect.* Sin embargo para desarrolladores de software "de base" como compiladores o en general aplicaciones que requieran información en tiempo de ejecución de clases y objetos, si será necesario usar las clases de este paquete directamente.

El "trabajo" de introspección lo realiza la clase ***Introspector*** del paquete *java.beans*, la que provee información sobre los eventos, propiedades y métodos de un bean.

Para cada una de estas características del bean, primero analiza en la clase y superclases si existe información explícita dejada en una clase BeanInfo.

En caso de no existir información explícita, el *introspector* buscará incluso información dejada de manera implícita :

A través de patrones de diseño para identificar los métodos de acceso a las propiedades (get... , set...), fuentes de eventos y métodos públicos.

En este último caso el introspector si utiliza servicios de reflexión de "bajo nivel".

La clase Class

Permite acceder a información de una clase en tiempo de ejecución. Las instancias de esta clase son "reflejos" de sus correspondientes clases o interfaces.

java.lang.reflect

La clase ***Field*** provee los métodos necesarios para acceder a los campos (variables de clase y de instancia) de una clase u objeto.

La clase ***Method*** provee los métodos necesarios para acceder a información de los métodos - abstractos, estáticos o de instancia - de una clase o interfaz.

Sólo la máquina virtual de Java puede crear instancias de las clases *Class*, *Field*, *Method*, *Constructor* y *Array* (sus constructores son privados).

Construcción de un bean

Sólo la etapa de planeamiento es necesariamente la primera. Las otras son generalmente realizadas en paralelo. Por ejemplo, aunque se parte de un conjunto "estimado" de propiedades, generalmente se adicionarán algunas más necesarias al diseñar la funcionalidad (métodos).

Etapas :

1.- Planeamiento
- 1.1 Establecer el objetivo del bean
  Prácticamente lo más importante pues definir precisamente lo que hará el bean es indispensable para un eficaz diseño del componente.
- 1.2 Pronosticar el ambiente
  Quiénes y cómo utilizarán el bean es una pregunta que redundará en un diseño que persiga un eficiente uso del mismo.
- 1.3 Pronosticar o prever nuevos requerimientos futuros
  Permitirá diseñar el bean de manera que quede flexible para agregarle funcionalidad futura.

2.- Diseño de propiedades
Establecer los posibles estados del bean ayudará a definir las propiedades y a su vez cuáles de estas deberán ser accesibles externamente.

3.- Definición de Métodos Públicos
Puede empezarse con aquellos necesarios para acceder (recuperar o fijar valores de) propiedades.
Definir los constructores es también un paso importante.

4.- Identificación y Definición de Eventos

Si bien los eventos (como en las aplicaciones de Java) son implementados a través de métodos (para registrar *listeners* y responder a las notificaciones de eventos), desde un punto de vista conceptual y de diseño pueden verse separadamente de éstos.
La funcionalidad y objetivos del bean determinarán qué tipos de eventos será capaz de originar. Estos eventos pueden ser estándares (de los paquetes de eventos de java : *java.awt.event* o *com.sun.java.swing.event)* o definidos por quien vaya a desarrollar el bean. En este último caso deberá implementarse una clase por cada tipo de evento no estándar. Para ambos casos deberán definirse los métodos para los listeners de cada tipo de evento.
También deberán definirse los eventos a manejar internamente en el bean.

Programación Visual

Como ya se vio, JavaBeans es el modelo de componentes estándar del lenguaje Java.

También se vio que la característica de introspección para los componentes bajo este modelo facilita su integración con herramientas visuales para desarrollo aplicaciones.

VisualAge for Java adopta el modelo JavaBeans para desarrollo con componentes. Asimismo, integra en su IDE las bibliotecas de beans ofrecidas por JavaSoft, de manera de usarlas en la construcción de aplicaciones, applets e incluso otros beans que, a su vez pueden ser integrados a la herramienta para su utilización.

Los beans visuales (para interfaces de usuario) que utiliza Visual Age están basados en las clases de los paquetes AWT y Swing de Java. Sin embargo el Editor de Composición Visual de VAJava es extensible, esto es que puede incorporar beans de otras fuentes y también aquellos desarrollados por el propio programador.

Los beans visuales se encuentran iconizados en la paleta y se les puede ubicar luego de seleccionar la correspondiente biblioteca (en la diapositiva pueden apreciarse íconos de las clases de componentes de la biblioteca swing.

Basándose en el paradigma de objetos y su característica de "comunicación por mensajes" para especificar la funcionalidad de una aplicación, el editor de composición visual ofrece una sofisticada pero intuitiva forma de especificar estos mensajes : Usa conectores para especificar los mensajes e interacción entre los componentes que pueden verse gráficamente.

Como hasta los componentes "no visuales" pueden verse, no existirá limitación para conexiones entre componentes. Entonces gran parte de la aplicación podrá "programarse" visualmente.

Las conexiones (o conectores), que son a su vez objetos, también permitirán al desarrollador agregar o integrar código en lenguaje Java a la aplicación.

Construcción de un applet con beans y eventos

Para apreciar la construcción de una aplicación (en este caso un applet) utilizando componentes de una biblioteca y haciendo uso de facilidades visuales del Visual Age for Java, se muestran los pasos para construir un applet cuya función será presentar un campo de texto y una lista de texto. Los items de texto que el usuario escriba en el campo de texto podrán pasarse a la lista mediante un clic de un botón con una etiqueta que indique esta función. Ítems de la lista de texto podrán eliminarse mediante clic en otro botón que indique esta función.

La herramienta Visual for Java (VAJava) facilita el inicio de la programación visual mediante un "SmartGuide" que permite especificar el nombre de la clase, la superclase de la que se extenderá y que se construirá de manera visual. También permite crear el proyecto y paquete en caso de especificar uno no existente.

El editor de composición visual es parte de la facilidad del navegador ("browser") de una clase y presenta un área libre donde podrán distribuirse los componentes (beans) que conformarán la GUI de la aplicación. Presenta también una paleta donde los beans están agrupados en categorías.

El área de estado informa sobre la categoría y bean seleccionado de la paleta o del componente o conector seleccionado en el área libre.

Arrastrando beans : definiendo la GUI de la aplicación

Desde la paleta se podrá arrastrar un bean hacia el área libre.
Para esto se hace clic sobre el ícono del bean en la paleta y se arrastra hasta el área libre.

En la diapositiva puede apreciarse que fueron colocados en el área libre (se arrastraron) los beans :

- Un campo de texto (bean : *JTextField*)
- Una etiqueta para este (el bean *JLabel*)
- Un campo Lista con capacidad de barrido (primero se colocó el bean *JScrollPane* de manera que los items puedan ser "barridos" en caso de requerir mayor espacio que su contenedor para su despliegue y luego el bean *JList* dentro del *JScrollPane*).
- Una etiqueta para esta lista (el bean *JLabel*)
- Dos botones (beans de la clase *JButton*)

Una vez que se ha arrastrado un ícono de la paleta y se ha colocado en el área libre, lo que en realidad se está haciendo es creando una instancia de la clase seleccionada.

Cada una de estas instancias tiene propiedades que, siendo parte de su interfaz, pueden ser modificadas. VAJava, a través de su editor de composición visual, ofrece un editor de estas propiedades para cada bean.

Para acceder al editor de propiedades se tienen varias alternativas :
a) Hacer doble clic sobre el bean (la instancia) cuyas propiedades quieren editarse
b) Seleccionar el menú *"Tools"* y luego la opción *"Properties"*
c) Hacer clic derecho sobre el bean y seleccionar la opción "*Properties*"
d) Hacer clic en el botón correspondiente (el segundo) de la barra de herramientas.

Las propiedades mostradas por el editor, evidentemente dependerán de la clase de bean seleccionado.

Las propiedades que para el ejemplo mostrado tuvieron que ser definitivamente modificadas fueron las de texto (*text*) para las etiquetas y los botones.

Incorporando beans no visuales

Es necesario hacer una precisión sobre la librería de clases de Java (JFC por sus siglas en inglés y que se encuentra como el proyecto "JFC class libraries" en Visual Age for Java) :

En estas clases, los datos y las vistas de los datos están en objetos separados.

Entonces para el ejemplo mostrado se requiere un objeto (bean) para guardar los items que irán en el objeto visible (*JList*).

El bean requerido es el *DefaultListModel* que es una clase que implementa el API de *Vector* (clase del paquete *java.util* que es un arreglo de objetos que puede crecer y cuyos elementos pueden ser accedidos por un índice). Este bean representa el contenido de la lista (*JList*) además de notificar los eventos de cambios a los "listeners" registrados.

Para incorporar beans no visuales el editor de composición visual de VAJava ofrece el botón de selección de beans (Choose Bean) de la paleta.

A su vez también permite - mediante una ventana apropiada - buscar el bean adecuado a las necesidades de la aplicación.

Conexiones Evento-a-Método

La funcionalidad requerida para la aplicación del ejemplo es que el texto contenido en el campo de texto pase a la Lista cuando el botón "Agregar" sea presionado (con un clic de mouse).

Entonces se requiere conexiones entre el botón, el campo de texto y la Lista. Sin embargo quien contiene los datos de la lista es el bean DefaultListModel, por lo que las conexiones deberán realizarse hacia este bean.

Hacer clic a un botón origina un evento *actionPerformed (ActionEvent evento).* Agregar un item a una lista se realiza a través del método *addElement(Object objeto).*

Entonces la conexión necesaria será entre el evento *actionPerformed(ActionEvent evento)* del botón "Agregar" y el método *addElement(Object objeto)* del bean DefaultListModel.

Para establecer las conexiones se hace uso de los menús que la herramienta tiene disponibles por cada componente existente en el área libre. Por ejemplo para la conexión entre el botón "Agregar" y el DefaultListModel será suficiente hacer clic derecho sobre el botón elegir la opción "Connect" y luego elegir "actionPerformed" para luego indicar el destino de la conexión : el ícono del bean no visible DefaultListModel y su respectivo método addElement(java.lang.Object)".

El hecho que la conexión aparezca discontinua, indicará que todavía no está completamente especificada. El motivo es que falta el parámetro del método addElement(java.lang.Object).

Para especificar la funcionalidad del otro botón ("Eliminar") se procede de forma similar pero entre el evento *actionPerformed(ActionEvent evento)* del botón "Eliminar" y el método *removeElementAt(int entero)* del bean DefaultListModel.

El color de este tipo de conexiones es verde.

Conexiones Parámetro-de-Propiedad

El hecho de haber incorporado estas conexiones evento-a-método no es suficiente para agregar o remover nada de la lista ya que tanto el método *addElement(Object)* como *removeElementAt(int)* requieren parámetros que especifiquen el objeto a agregarse o eliminarse de la lista.

Las conexiones parámetro-de-propiedad permiten especificar gráficamente qué propiedad (y de qué objeto) se usará como parámetro en una conexión.

Esto se hace conectando el objeto, una de cuyas propiedades se pasará como parámetro, con la conexión (flecha) relacionada con el método que requiere el parámetro.

En el caso del ejemplo, el elemento a agregar a la lista y por lo tanto el parámetro a pasar en el método *addElement* es el texto (propiedad) del bean JTextField (objeto).

De igual manera, el elemento que está en un momento seleccionado del bean JList (la lista) será el parámetro para el método *removeElementAt* para eliminar un elemento de la lista.

El color de este tipo de conexiones es rojo.

Conexiones Fuente-Vista (de datos)

Como ya se mencionó, para componentes que contienen colecciones de datos, los datos y las vistas de los datos están en objetos separados.

Cuando se está programando visualmente una aplicación, también se requerirá especificar mediante una conexión, la funcionalidad para enviar los datos del objeto que los contiene (la **fuente**) hacia el objeto que los muestra (la **vista**).

En el caso del ejemplo será necesario conectar la fuente (el modelo DefaultListModel) con la vista (la lista).

El color de este tipo de conexiones es azul.

Conexiones Evento-a-Código

Una conexión Evento-a-Código permite asociar un bloque de código, escrito en lenguaje Java, con un evento.

Aunque para el ejemplo mostrado, bastará con las especificaciones gráficas dadas con las conexiones para que VAJava genere el código correspondiente para el applet, en ocasiones será necesario incorporar lógica adicional.

VAJava provee la facilidad de esta asociación entre un evento y métodos codificados por el desarrollador además de incorporarlos al editor de composición visual.

En el menú que VAJava provee para cada componente del área libre aparecerá la opción "Event to code", la que al elegirla invoca a la ventana para edición del código.

En la diapositiva se muestra un enlace entre el evento clic al botón "Agregar" y un método llamado jButton1_ActionEvents()

El color de este tipo de conexiones es verde pues es en la práctica una conexión evento-a-método.

Bases de datos

Una base de datos es un conjunto de datos organizados de tal manera que pueda extraerse información.
La organización de los datos persigue el objetivo que estos puedan ser compartidos por varios usuarios.

Sistema de administración de base de datos (DBMS)

DBMS por sus siglas en inglés (DataBase Management System).

Software que administra el acceso a los datos, permitiendo su almacenamiento, consulta y actualización. Tiene la capacidad de responder a múltiples usuarios accediendo en forma concurrente a los datos.
Provee facilidades para la administración del conjunto como toma de respaldos, recuperación.

El DBMS permite tener los datos de toda la organización (incluida la información de sus principales entidades) de forma integrada, de manera que estos se encuentren disponibles a consultas o actualizaciones de transacciones realizadas por el personal de la empresa, clientes de la misma, a través de aplicaciones de un sistema de información o directamente a través de un lenguaje que sea "comprendido" por el DBMS. Este lenguaje, en el caso de bases de datos relacionales (RDBMS) es el SQL, lenguaje que no sólo permite la comunicación para acceder los datos sino para también permite definirlos.

Actualmente los software RDBMS se encuentran en diversas plataformas, aunque son desarrollados para arquitecturas tanto mainframe como cliente/servidor corriendo en el "back-end" (host o servidor) usando la infraestructura de red disponible para la comunicación con los usuarios.

Plataforma Cliente/Servidor para uso de DBMS's

Cliente/Servidor implica distribución de aplicaciones y/o datos en una red de computadoras. Conocida por sus siglas C/S, también es sinónimo de computación abierta que permite utilización de hardware y software variado, sin dependencia de un sólo proveedor.

La plataforma cliente/servidor tiene tres componentes :

En el **cliente** corre la parte de la aplicación correspondiente. Lo hace en el sistema operativo, que a su vez provee la interfaz usuario (U.I.), hace ya buen tiempo gráfica (GUI) u orientada a objetos (OOUI), y puede acceder diferentes servicios distribuidos. Para acceder a servicios distribuidos lo hace a través del componente middleware, quien maneja los servicios que no son locales.
En el cliente también corre un componente del middleware de administración de sistemas distribuidos (DSM)

El componente de aplicación en el **servidor** generalmente corre sobre un software para la correspondiente plataforma (del servidor) : un DBMS con SQL (el caso que compete a las bases de datos), un monitor de transacciones, groupware, servidores de objetos y el Web.
Depende del S.O. para "interfazar" con el componente middleware que hace los requerimientos de servicios. También corre un componente del DSM.

El componente **middleware** corre tanto en el cliente como en el servidor.
Se puede clasificar en tres categorías de middleware :

1.- El software (o mejor dicho los protocolos) de transporte.
Provee comunicación a través de WANs y LANs y por supuesto la necesaria combinación LAN/WAN/LAN.
Estos protocolos vienen como drivers en los sistemas operativos modernos, los que proveen interfaces muy bien definidas entre componentes de manera de llegar desde las aplicaciones hasta los adaptadores de red.
2.- El sistema operativo de red
Aunque el término "de red" ya prácticamente quedó en el pasado, pues los sistemas operativos ofrecen la funcionalidad correspondiente, son las funciones "de red" las que principalmente son usadas en un ambiente cliente servidor :
Servicios de directorio, compartición de recursos, seguridad, etc.
Facilidades para internet/intranet son proporcionadas también por los sistemas operativos.
3.- Servicios específicos
También deben correr en ambos lados, cliente y servidor, de manera de proveer la funcionalidad necesaria como por ejemplo acceso y recuperación de datos de una base de datos, correo electrónico, brokers de objetos y otros.

El Modelo Relacional para Bases de Datos

Modelo de datos basado en dos teorías matemáticas : La teoría de relaciones y la lógica de predicados de primer orden. Fue presentado por E.F. Codd cuando trabajaba para la IBM, en el año 1970. La versión de 1990 presenta 333 reglas, presentando al modelo como un producto completo de donde contrastar un desarrollo de software RDBMS. El modelo relacional presenta una estructura, operaciones y reglas de integridad.

Estructura

Los datos son agrupados como relaciones de datos y se representan en forma de tablas, donde cada fila corresponde a una tupla de la relación y cada columna representa a un dominio.

Dominio.- Conjunto de valores que puede tomar un dato.

No existe otra forma de agrupar datos.
Cada dato es atómico, representa un solo valor del dominio

Cada columna representa diferente información.
Aunque dos columnas de una misma relación puedan tener exactamente el mismo dominio, representarán información diferente. Cada columna es un USO PARTICULAR de un dominio.

Operaciones Relacionales

Restricción o Selección
Operación unitaria que produce una relación con tuplas seleccionadas de la relación operando (R) que satisfagan un predicado (p). En el Predicado los operandos pueden ser columnas o constantes. Los operadores pueden ser de comparación (=, ¬=, < , <= , > , >=), aritméticos (+ , - , * , / , **) o lógicos (Y, O , ¬).

Proyección
Operación unitaria que produce una relación con las columnas de la relación operando (R) que se den como argumento (A) y eliminando de este resultado las tuplas repetidas.
El argumento es una lista de columnas : A = (a1, a2, ..., aN)

Producto Cartesiano
Operación binaria (o mayor) que produce una relación que contiene como tuplas a todas las combinaciones de las tuplas de la primera relación operando (R1) con las de la segunda (R2). La tabla resultado tiene como columnas la concatenación de las columnas de ambas.

Unión
Operación binaria (o mayor) que produce una relación que contiene todas las tuplas de la primera relación operando (R1) y todas las de la segunda (R2) incluyendo sólo una vez aquellas comunes.
Las relaciones operandos deben ser "compatibles para unirse" : ser del mismo grado y los dominios de las columnas (por lo menos de una) de ambas relaciones deben ser iguales. La unión entre relaciones es parecida a la unión de conjuntos, pero con restricciones sobre los conjuntos que intervienen.

Diferencia
Operación binaria que produce una relación que contiene todas las tuplas de la primera relación operando (R1) que no se encuentren en la segunda (R2).

Intersección
Operación binaria (o mayor) que produce una relación que contiene todas las tuplas de la primera relación operando (R1) que también están en la segunda relación operando (R2).

Conjunción o Join Relacional
Operación binaria (o mayor) que produce una relación que contiene como tuplas aquellas combinaciones de las tuplas de la primera relación operando (R1) con las de la segunda (R2) que satisfagan un predicado.
Es una operación derivada pues puede remplazarse por una selección aplicada a un producto cartesiano.

El modelo Relacional también provee operaciones para actualización de datos y tuplas y relaciones : **Asignación**, **Inserción**, **Eliminación** y **Actualización**.

Diseño Lógico de Bases de Datos

El diseño de una Base de datos es parte fundamental del proceso de desarrollo de sistemas de información. De una manera general pueden identificarse dos etapas en el diseño de Bases de datos : El diseño lógico y el diseño físico.

El diseño lógico es independiente de la tecnología particular que vaya a utilizarse y utiliza como principal técnica al modelamiento de datos.
Cabe mencionar que si es que la organización ha realizado un planeamiento estratégico (paso necesario previo a un planeamiento de sistemas que incluya bases de datos) podrá utilizarse importante información de éste como las Entidades críticas de la organización.

El diseño físico es la implementación del diseño lógico sobre una tecnología de base de datos particular.

Modelamiento : Relaciones y Cardinalidades

El modelamiento de datos es la principal técnica para diseño lógico de bases de datos y consiste en reconocer las entidades reales (del sistema real, por ejemplo una organización) y convertirlas en entidades de datos que incluyen sus características como atributos (datos).

En un modelo de datos las entidades se representan en rectángulos y los atributos de las mismas dentro de estos.
Otra característica fundamental del modelamiento de datos es que contempla la semántica de las relaciones entre entidades y reglas de negocio válidas para el sistema real. Las relaciones se representan por líneas que unen a las entidades y las reglas por las cardinalidades de estas relaciones.

En la diapositiva puede apreciarse una de las notaciones más usadas para especificar relaciones y cardinalidades. La mayoría de herramientas CASE que apoyan el proceso de modelamiento también permiten especificar y generar el modelo lógico de acuerdo a un DBMS.

Integridad referencial y reglas de integridad referencial

La integridad referencial permite definir relaciones entre tablas a través de los valores de los datos (columnas) de estas tablas.

El DBMS "mantiene" estas relaciones a través de "restricciones referenciales" que especifican que todos los valores de una determinada columna de una tabla, también existan en otra columna de la misma u otra tabla.
El caso más usado es el que una columna de una determinada tabla (HIJO) se refiere a la llave primaria de otra tabla (PADRE). Esta columna se dice llave foránea.
Una restricción de integridad referencial controlada por el DBMS, verificará que efectivamente el valor que se intenta registrar (INSERT) o actualizar (UPDATE) en una columna de una tabla "hijo" exista y corresponda a uno de los valores de la respectiva columna (única y por lo general llave primaria) en la tabla "padre" o en todo caso se permita un valor NULL si así fue definida. En este último caso, en el que se permite valor NULL en la columna de la tabla "hijo" o llave foránea, se dice que la relación referencial es "condicional" y gráficamente se expresa con la correspondiente cardinalidad mínima de CERO.

Restricciones y Acciones de integridad referencial

Los DBMSs permiten definir restricciones o acciones a efectuar ante una acción sobre filas de la tabla padre.

Ante una acción de actualización (como una sentencia UPDATE o DELETE) sobre una fila de una tabla padre, se definen acciones sobre las correspondientes filas en la tabla hijo (relacionadas por los mismos valores en la llave foránea) o también puede restringirse esta acción en la tabla padre.

Por ejemplo puede especificarse restringir el borrado de una fila de una tabla padre en caso que existan filas "hijo" correspondientes. Esto es una opción RESTRICT ante un DELETE en la tabla padre y se especifica en la sentencia SQL de definición de datos (DDL) CREATE TABLE mediante la opción ON DELETE RESTRICT en la cláusula REFERENCES.

Esta especificación, hará que el DBMS, antes de proceder a eliminar una fila en una tabla padre, verifique que no existan filas con el mismo valor en sus llaves foráneas que el valor de la llave de la fila que desea eliminarse. En caso existiera una o más filas, no se efectuará la eliminación y se enviará el mensaje correspondiente.

La opción CASCADE especificará que ante una eliminación (DELETE) de una fila en la tabla referenciada (padre), también se eliminen las filas referenciadas (que contengan en la llave foránea el mismo valor de la llave padre que se está eliminando).

Para relaciones referenciales condicionales :

La opción SET NULL especificará que ante una eliminación (DELETE) de una fila en la tabla referenciada (padre), las filas referenciadas (que contengan en la llave foránea el mismo valor de la llave padre que se está eliminando) pasarán a contener el valor NULL.

SQL

Actualmente es el protocolo para bases de datos para la mayoría de plataformas y proporciona diversas facilidades, tanto para la interacción con bases de datos, administración de las mismas y el desarrollo de sistemas de información con bases de datos :

Es un lenguaje interactivo que permite realizar consultas ad-hoc a bases de datos.
Puede ser embebido en algún otro lenguaje como C, C++, COBOL, etc.
SQL es una lenguaje no sólo para consultas sino para definición y administración de bases de datos.
SQL también provee de las facilidades para el control a los datos par parte de varios usuarios en un ambiente de transacciones concurrentes.

SQL : Lenguaje de definición de datos (DDL)

El SQL provee un conjunto de comandos que permiten definir las estructuras (objetos de BD como tablas, vistas, índices, tipos de datos) del diseño de la base de datos.

Los principales son los que permiten crear, borrar y alterar características de los objetos de base de datos : CREATE, DROP y ALTER.

Los tipos de datos SQL caracterizan a los DBMSs

Sólo algunos sistemas DBMS ofrecen tipos de datos definidos por el usuario.

Diseño Físico de bases de datos - tipos de datos

Tal como se mencionó, el diseño físico es la implementación del diseño lógico sobre una tecnología de base de datos particular.

El diseño físico consiste en seleccionar el tipo de datos ofrecido por un DBMS particular para cada atributo de las entidades de datos del modelo de datos.

Asimismo, se definen las reglas de integridad y acciones de integridad referencial haciendo uso de los comandos de definición del SQL.

SQL : Lenguaje de manipulación de datos (DML)

Parte del SQL usado para extraer datos o actualizarlos en la base de datos.
Generalmente no existe una forma única para realizar una operación sobre datos, sobretodo en caso de consultas.
Las cuatro sentencias principales de manejo de datos son : SELECT, INSERT, UPDATE, DELETE.

SELECT

La sentencia SELECT implementa básicamente a la operación de Join Relacional, que es una operación que puede verse como una selección (o restricción) sobre un producto cartesiano. A esto se le incorpora la facilidad de elección de columnas a mostrar en el resultado (también ofrece la facilidad de la operación relacional de proyección).

La expresión condicional, que corresponde al "predicado" de la operación relacional de selección (restricción), debe colocarse a continuación de la palabra reservada WHERE, la que acepta varios operadores de comparación :

=	igualdad
<>	desigualdad
<	menor
<=	menor o igual
>	mayor
>=	mayor o igual

Asimismo permite usar operadores lógicos ("booleanos"):

AND	Y lógico
OR	O lógico

LA cláusula WHERE no es obligatoria, sin embargo el no colocarla implicará una operación de producto cartesiano, la que resultará en todas las combinaciones de las filas de las tablas y vistas especificadas en la lista FROM.

La sentencia SQL SELECT ofrece la cláusula ORDER BY para ordenar el resultado final.

También permite agrupar/agregar las filas del resultado de la restricción haciendo uso de la cláusula GROUP BY y las funciones de columna ofrecidas por el DBMS.

Al usar GROUP BY puede restringirse aún más el resultado haciendo uso de la cláusula HAVING.

INSERT

La sentencia INSERT inserta filas en una tabla o vista.

La inserción de una fila en una vista también inserta la fila en la tabla en la que se basa la vista.

Tal como puede apreciarse en la sintaxis de la sentencia INSERT, pueden insertarse filas a una tabla, especificando explícitamente los valores mediante la cláusula VALUES o a través de una selección con una sentencia SELECT.

El DBMS no sólo verifica la correcta sintaxis de la sentencia sino que verifica la integridad de la base de datos respecto a los valores que se están tratando de registrar.

Por ejemplo no permitirá que se inserte una fila que contenga un dato de llave primaria con un valor igual al de la llave de una fila existente.

De la misma manera, el DBMS verificará cualquier otra regla de integridad definida a través de sentencias DDL de SQL o a través de activadores (triggers).

Puede indicarse el valor nulo NULL para aquellas columnas que permitan este valor (aquellas no definidas con NOT NULL en la sentencia CREATE TABLE).
También puede indicarse la palabra reservada DEFAULT en la lista VALUES para indicar que se requiere insertar el valor definido como por omisión (DEFAULT especificado en la sentencia CREATE TABLE).

Dependiendo de la definición de cada columna (en el CREATE TABLE) el valor por omisión a insertarse será uno del sistema, el proporcionado por el usuario (DEFAULT en CREATE TABLE) o un valor nulo.

Si se omite un valor de la lista VALUES para una columna que había sido definida con la opción NOT NULL pero sin la opción WITH DEFAULT, se recibirá un mensaje de error. Esto es debido a que las columnas que no usan WITH DEFAULT tienen como valor por omisión e valor nulo.

La sentencia INSERT es muy práctica para insertar datos a una tabla, sin embargo no es recomendable para cantidades voluminosas de filas a insertar por las consecuencias de sobrecarga en el historial de transacciones.
El DB2 provee dos facilidades para cargas voluminosas : IMPORT y LOAD.

DELETE

Existen muchas formas de eliminar datos de una base de datos.

Por ejemplo pueden eliminarse tablas enteras mediante la sentencia DROP TABLE o espacios de tabla mediante la sentencia DROP TABLESPACE.
Incluso - con el correspondiente nivel de autorización por supuesto - puede eliminarse la totalidad de una base de datos : DROP DATABASE.

Para eliminar una o un grupo de filas de una tabla, existe la sentencia SQL DELETE.

A diferencia de la sentencia SELECT (donde puede obtenerse ciertas columnas) con la sentencia DELETE se eliminan filas completas.
No es posible eliminar algunas columnas.

Existen dos formas de esta sentencia:

1. La forma DELETE con búsqueda

Se utiliza para suprimir una o varias filas (determinadas opcionalmente mediante una condición de búsqueda).

2. La forma DELETE con posición

Se utiliza para suprimir una fila exactamente (determinada por la posición del cursor en un determinado momento) :

La sintaxis de ambas formas se muestra en la diapositiva.

La sentencia DELETE es muy práctica para eliminar datos a una tabla, sin embargo no es recomendable para cantidades voluminosas de filas a eliminar por las consecuencias de sobre carga en el historial de transacciones.

UPDATE

La sentencia UPDATE permite la actualización de una o más columnas de una, algunas filas (que satisfagan una condición especificada) o todas las filas de una tabla o vista.

La actualización de una fila de una vista actualiza una fila de su tabla base.

Las formas de esta sentencia son:

1. La forma UPDATE Con búsqueda

Se utiliza para actualizar una o varias filas (determinadas opcionalmente por la condición de búsqueda).

Si no se especifica la condición de búsqueda (cláusula WHERE) todas las filas de la tabla serán actualizadas.
La sintaxis de esta forma es la mostrada en la diapositiva.

2. La forma de UPDATE Con posición se utiliza para actualizar exactamente una fila (tal como determina la posición de un cursor en un determinado momento) :

```
>>-UPDATE---+-nombre-tabla-+-SET--| cláusula-asignación |------->
            +-nombre-vista-+

>--WHERE CURRENT OF--nombre-cursor----------------------------><
```

SQL : Lenguaje de control (DCL)

El SQL es un lenguaje que también provee comandos para control sobre la base de datos, para funciones como las de conexión a la base de datos y otorgamiento de privilegios a usuarios o grupos para acceder a objetos de base de datos.

Control de Acceso a los datos

Una de las funciones y facilidades que proveen los DBMS es la que se refiere al control del acceso a los datos de manera de proveer un ambiente de seguridad, tanto para el desarrollo de los sistemas de información, como para es uso permanente de los mismos. Existen tres factores para este control :

1. Quién accederá a los datos ?
Los usuarios de los datos tienen que ser conocidos por el DBMS.

2. Qué es lo que será accedido ?
Los datos son almacenados en diferentes tipos de objetos de base de datos, como tablas, vistas, índices, columnas, etc.

3. Qué tipo de acceso será permitido ?
Existen diversas acciones que pueden realizarse sobre los objetos de base de datos, tales como, crear, actualizar, borrar, etc.

Niveles de seguridad

Existen tres niveles de seguridad para el control del acceso a una base de datos.

Un primer nivel controla el acceso a la instancia. El segundo nivel controla el acceso a la base de datos. Es el tercer nivel el que tiene que ver con el acceso a objetos de base de datos específicos en la base de datos.

El primer nivel es manejado por las facilidades de seguridad provistas por la plataforma donde corre el DBMS. Estas facilidades externas pueden ser parte del S.O. o un producto aparte y se encargan de la verificación que la persona usuaria sea efectivamente quien dice ser. Además también controlan el acceso a aplicaciones ubicadas en directorios. Este proceso llamado autenticación, generalmente consta de proveer un nombre de usuario y una contraseña.
Los siguientes dos niveles si son controlados directamente por el DB2.
Los niveles de autorizaciones proveen una jerarquía de capacidades de administración.
Los privilegios son los "derechos" para crear o acceder a un objeto de base de datos.

Privilegios sobre objetos de base de datos

Un privilegio es el "derecho" que posee un usuario en particular o un grupo de usuarios para crear o acceder a un objeto de base de datos.

Los privilegios son mantenidos por el DBMS en el catálogo del sistema, precisamente para el control a los recursos correspondientes.

Existen tres tipos de privilegios :

1.- Privilegio de propiedad o de control.

Para la generalidad de objetos de base de datos, el usuario que lo crea tiene acceso completo a éste. Las vistas son la excepción a esta regla.
"Tener el control" de un objeto significa poder accederlo, dar permiso a otros para accederlo y por último dar permiso a otros usuarios a que estos puedan también otorgar permisos.

Los privilegios son otorgados o retirados por los usuarios "dueños" de los objetos o por usuarios con autorización de administración a través de las sentencias **GRANT** y **REVOKE** respectivamente.

2.- Privilegio Individual

Permite a un usuario realizar una acción específica sobre todos o - a veces - algún objeto específico.
Este tipo de privilegio involucra las acciones como SELECT, DELETE, INSERT.

3.- Privilegio Implícito

Es aquel privilegio que un usuario obtiene indirectamente al encontrarse ejecutando un proceso.
Por ejemplo cuando un usuario ejecuta un "paquete", el cual involucra una serie de privilegios, este usuario obtiene esos privilegios mientras ejecuta el paquete. Esto es conocido como privilegio EXECUTE.

Control de transacciones sobre bases de datos

Una de las responsabilidades de los sistemas DBMS es mantener la base de datos íntegra, esto es en un estado coherente de información.

Como estos software corren sobre sistemas físicos expuestos a diversos fenómenos como accidentes, apagones, sabotaje o simples fallas de hardware, proveen mecanismos para asegurar los datos.

Una de las maneras de asegurar la integridad de los datos es agrupando operaciones en transacciones, de manera que el éxito o fracaso sea de todo el conjunto y no de una operación particular.

Las sentencias de control del SQL COMMIT y ROLLBACK actúan en las transacciones y comprometen definitivamente con las operaciones a la base de datos (COMMIT) o dejan todo en el estado inicial de la transacción, como si nada hubiese ocurrido (ROLLBACK).

COMMIT
Concluye la transacción y "compromete" la base de datos haciendo permanente los cambios indicados a través de las sentencias SQL correspondientes.

ROLLBACK
Revierte las acciones relativas a los cambios indicados a través de las sentencias SQL, esto es que la base de datos no queda "comprometida".

Ambas sentencias liberan el bloqueo a los recursos que pudieran haberse efectuado por esas mismas sentencia.

Necesidad de Bloqueo de recursos

El bloqueo de recursos tiene por objetivo evitar las anomalías o fenómenos que tienen posibilidad de producirse en un ambiente concurrente (donde múltiples usuarios acceden al mismo tiempo a los mismos recursos de datos).

Por ejemplo para no permitir el acceso a datos no comprometidos, que es la causa que pueda producirse una lectura falsa o una actualización perdida.

Niveles de Aislamiento

El aislamiento de transacciones es el grado de inacción entre transacciones concurrentes.

Los niveles de aislamiento son una medida en que el aislamiento de transacciones tiene éxito o - visto desde un punto de vista opuesto - el grado de interferencia que una transacción permitirá a otras transacciones concurrentes a ella.

El DB2 UDB provee varios de estos "niveles de protección" cumpliendo con el estándar SQL-92 que define cuatro niveles de aislamiento (o estrategias de bloqueo). En el cuadro se puede apreciar los cuatro niveles de aislamiento del DB2 UDB en términos de la permisividad de los fenómenos.

A mayor nivel de aislamiento, menor nivel de interferencia entre transacciones, pero a su vez, menor concurrencia.

El mayor nivel de aislamiento del DB2 UDB permite un ambiente de transacciones completamente "serializable".

El nivel de aislamiento se especifica al momento de preparar un programa que contiene SQL embebido.
En caso de no especificarse, el valor asumido por omisión es CURSOR STABILITY

java.sql : el paquete JDBC

La especificación Java DataBase Connectivity, más conocida por sus siglas JDBC, constituye una interfaz para programar aplicaciones (API) con el objetivo de acceso a bases de datos relacionales.

Como está escrita en Java, entonces obviamente constituye un conjunto de clases e interfaces. Estas están agrupadas en el paquete java.sql.

JDBC está implementado en dos niveles :

1.- El nivel de driver, que consiste en implementaciones específicas de interfaces (provistas por el desarrollador del driver) que trabajan con un manejador de base de datos relacional (RDBMS) específico.

2.- El nivel de la aplicación, que consiste en la clase DriverManager que maneja los drivers siendo utilizados por una aplicación o applet para acceder una base de datos en particular.

Esta abstracción de un DBMS no sólo permite conectarse y acceder datos de bases de datos heterogéneas al mismo tiempo y usando los mismos métodos, sino que encapsula (escondiéndolos del desarrollador) los detalles de implementación (sockets, streams de E/S, etc.).

La idea es que el desarrollador, a través de simples llamadas a métodos pueda :

- Crear una conexión a base de datos
- Ejecutar una consulta (query)
- Manejar los resultados de consultas
- Etc.

La clase DriverManager

La clase DriverManager controla la carga de clases de drivers específicos y provee el servicio de manejar este conjunto de drivers JDBC.
Esta clase primero trata de cargar un driver por omisión, que es el referido en la propiedad del sistema *jdbc.drivers*.

Los drivers también pueden ser cargados dinámicamente en tiempo de ejecución utilizando el método estático *forName* de la clase *Class*.

Ejemplo

```
Class.forName("com.ibm.jdbc.JdbcOdbcDriver")
```

Cuando el método getConnection es invocado DriverManager cargará los drivers según lo especificado.

La clase Types

La clase Types define constantes para identificar a los tipos de datos de SQL.

Estas constantes son usadas para hacer corresponder los tipos de SQL con tipos de datos Java.

Tal como se puede apreciar en la diapositiva, algunos tipos SQL no tienen una correspondencia con un tipo primitivo de Java sino que corresponden a uno compuesto : una clase (se ha especificado el paquete al que pertenecen).

JDBC categoría 1

JavaSoft ha desarrollado un puente (bridge) entre JDBC y ODBC. Ambas especificaciones están basadas en el estándar X/Open, que es una interfaz (API) a nivel de comandos SQL.

Este tipo de driver usa el driver ODBC específico para la base de datos que se desea acceder.

Requiere que el driver ODBC se encuentre disponible en la máquina cliente donde está corriendo la aplicación Java.

JDBC categoría 2

Este tipo de driver utiliza Java para hacer llamadas al API de acceso (protocolo) del DBMS que debe estar disponible en el cliente. Este último es quien provee la conectividad y acceso a la base de datos.

Entonces el driver es una implementación parcial en Java y depende que el ejecutable (código binario) del API al DBMS se encuentre en el cliente

Por ejemplo IBM, provee el Client Application Enabler (CAE) para acceder al motor relacional DB2 UDB.

JDBC categoría 3

Este tipo de driver utiliza los protocolos de red que vienen con el JDK (Kit de desarrollo de Java) para conectarse a un servidor. En este último se traducen los requerimientos a transacciones específicas del DBMS.

Esta forma no requiere la existencia de código ejecutable en el lado del cliente, pero implica la necesidad de un componente middleware para procesar las transacciones específicas.

Todas las funciones que no son especificadas en Java son de responsabilidad del middleware.

El protocolo genérico de red es flexible pues no requiere código instalado en el cliente, es así que con el mismo driver puede accederse a diferentes bases de datos (un ambiente heterogéneo).

JDBC categoría 4

Este tipo de driver está enteramente escrito en Java y se comunica directamente a la Base de datos con el protocolo de red del DBMS.

Esta comunicación directa se implementa a través de conexiones de red ("sockets").

Estructura de una aplicación JDBC

Antes de conectarse a una base de datos debe cargarse el driver JDBC.

Luego de conectarse con la base de datos puede crearse una instancia de clase Statement. Este objeto es el utilizado para representar la sentencia SQL, la misma que puede ser de tres tipos :

- Statement
Sentencia SQL estática, definida en el programa y sin variables host

- Prepared Statement
Sentencia SQL dinámica, que contiene variables host y/o partes variables a definirse en tiempo de ejecución.

- Callable Statement
Llamada a un procedimiento almacenado en el DBMS.

Al ejecutar la sentencia (utilizando el objeto *Statement*), el resultado se recibe en un objeto tipo *ResultSet*, que será quien contenga las filas del resultado de la sentencia (generalmente un SELECT).

Para procesar el conjunto resultado se utilizan los métodos del objeto *ResultSet.*

Generalmente será necesario "barrer" o moverse en el resultado. Para esto, se utiliza el método *next()* para avanzar de fila.

Cargar driver JDBC y conectarse a una Base de datos

En el ejemplo de la diapositiva se utiliza un driver categoría 2 para acceder a una base de datos local :

```
COM.ibm.db2.jdbc.app.DB2Driver
```

Como ya se mencionó, en esta categoría de driver, el API de acceso al DBMS debe estar en el cliente, entonces para el caso del ejemplo debe estar instalado el CAE de DB2 (driver nativo de IBM)

Para conectarse a una base de datos remota DB2 provee un protocolo genérico de red (driver de categoría 3) :

```
COM.ibm.db2.jdbc.net.DB2Driver
```

Pero además debe estar arrancado un JDBC server daemon (en la máquina servidor) y al momento de indicar la fuente de datos, la ubicación del recurso debe especificarse como un URL JDBC, de la siguiente forma :

```
jdbc:db2://IPhost:puerto/basededatos
```

por ejemplo :

```
jdbc:db2://Neptuno:8888/sample
```

db2jstrt 50000
6790

Ejecutar sentencia SQL y manejar resultado

Cuando una aplicación se conecta a una base de datos a través de JDBC crea un objeto tipo (que implementa la interfaz) *Connection*. Usando este objeto de tipo "Conexión" puede crearse un objeto tipo Statement (Sentencia SQL).

La ejecución de la sentencia da como resultado un objeto de tipo *ResultSet*.
el objeto *ResultSet* mantiene un cursor que inicialmente se pocisiona en la 1ra fila.
El método *next()* - invocado en resultados - avanza a la siguiente fila del resultado.

Invocando a métodos de objetos tipo *ResultSetMetaData* puede analizarse los tipos y propiedades de las columnas en un resultado (objeto tipo *ResultSet*).

En el ejemplo de la diapositiva se utiliza el método *getColumnCount()* para obtener el número de columnas del resultado (se almacena en la variable *columnas*).

El método *getColumnDisplaySize* obtiene el tamaño (caracteres) de la columna cuya posición es pasada como parámetro (*n*).

El método *getString* devuelve el valor de la columna *n* en la fila donde se encuentra el cursor. Lo devuelve como un *StringBuffer* de Java.

Se manejan las excepciones *InstantiationException* y *IllegalAccessException* pues podrían ser originadas por el método *newInstance*.

PreparedStatement : Sentencia para SQL dinámico

Las variables host son denotadas por signos de interrogación (?) en el texto de la sentencia SQL.

SQL dinámico se utiliza en casos que se requiera ejecutar una sentencia SQL varias veces pero con diferentes valores para condiciones (como en el ejemplo mostrado en la diapositiva) o cuando se requiere terminar la especificación de la sentencia en tiempo de ejecución (por ejemplo aceptando un valor ingresado por el usuario de la aplicación durante la ejecución para con ése valor "armar" la condición de una búsqueda).

Antes de ejecutar sentencias con SQL dinámico (preparadas) debe asignarse los valores a las variables host incluidas en ellas.

Para esto se utiliza el método *setTTTT(i, valor)* donde

TTT es el tipo Java correspondiente al tipo de datos válido en la sentencia SQL
i es la posición o número de variable host
valor es el valor a asignar a la variable

En el caso del ejemplo mostrado i=1 pues sólo hay una variable host.

Los Access Builders de VAJava

Las principales facilidades que ofrece la versión Enterprise de Visual Age for Java son los Access Builders:

DATA Access Builder
Permite acceder desde programas Java datos corporativos almacenados y manejados por motores relacionales (DBMSs) como por ejemplo el poderoso DB2 de IBM, a través de drivers JDBC.

RMI Access Builder
Hace uso de la especificación para invocación remota de métodos RMI para implementación de aplicaciones Java distribuidas.

CICS Access Builder
Facilita la construcción de programas Java con llamadas externas ECI (siglas en inglés de External Call Interfaces) a transacciones administradas por el monitor de transacciones CICS (Transaction Server)

C++ Access Builder
Facilita la construcción de aplicaciones distribuidas que se conectan con servidores de aplicaciones C++.

En términos generales un Access Builder analiza la especificación del servidor y construye beans que encapsulan la interfaz pública del mismo. Estos beans pueden utilizarse desde el editor de composición visual de VAJava conectándolos con componentes visuales (beans para GUI de la paleta).

En la diapositiva puede apreciarse que los Access Builder para DATA, RMI y CICS están completamente integrados al Workbench de VAJava.

El de C++ en cambio debe compilarse y el fuente Java debe integrarse al Workbwench a través de la facilidad de importación.

JDBC y las formas de utilizarlo

La especificación Java DataBase Connectivity, más conocida por sus siglas JDBC, constituye una interfaz para programar aplicaciones (API) con el objetivo de acceso a bases de datos relacionales.

Como está escrita en Java, entonces obviamente constituirá un conjunto de clases e interfaces.

Las configuraciones posibles para utilizar implementaciones de JDBC son :

1.- Un puente (bridge) entre JDBC y ODBC que traduzca las llamadas a JDBC a llamadas ODBC (que hagan lo mismo), las que son enviadas al DBMS.
La mayoría de DBMSs contemplan drivers para la especificación ODBC.

2.- Un driver JDBC que pasa el requerimiento al DBMS, cuyo SW cliente debe estar corriendo en la misma estación (PC) donde corre la aplicación Java.
Esta es la configuración genérica para correr aplicaciones del servidor.

3.- Un cliente JDBC que pasa requerimientos a un driver remoto (en otra locación) y al DBMS.
Esta configuración implica un cliente delgado (thin client) que no requiere código del DBMS en el front-end (máquina cliente).

Esta es la mejor configuración para applets pues el pequeño código JDBC puede cargarse de la red junto con el applet.

4.- Un driver JDBC totalmente desarrollado en Java
(no se muestra en la diapositiva)

El Data Access Builder

El Data Access Builder está basado en JDBC, que ya es un estándar y prácticamente todos los vendedores importantes de motores de bases de datos (DBMSs) lo incorporan en sus productos.

El Data Access Builder encapsula el código JDBC en beans.
El programa cliente accede a los beans sin el detalle de la codificación JDBC ni las características de la base de datos específica.

El Data Access Builder analiza las definiciones de las tablas y vistas en el catálogo del DBMS y genera beans de Java que encapsulan el acceso a las tablas en métodos de Java : las sentencias SQL SELECT, INSERT, UPDATE y DELETE son provistas como métodos (parte de la interfaz) de los beans generados.

DAB también soporta sentencias definidas por el usuario (p.e. joins), procedimientos almacenados, y operaciones del manejador como CONNECT, DISCONNET y COMMIT.

Los beans generados son integrados en el repositorio de VAJava y pueden ser usado desde el editor de composición visual.

Las principales funciones de los beans generados :

- Acceso a una sola fila (recuperación, actualización, inserción y eliminación).

- Recuperación de múltiples filas

- Operaciones del manejador (CONNECT, DISCONNET y COMMIT ROLLBACK)

- Componentes visuales que pueden ser utilizados en la GUI

- Una aplicación completa para acceder y manipular un subconjunto de la base de datos.

Desarrollo de Aplicaciones con Data Access Builder

Desde un punto de vista muy general puede mencionarse que los pasos necesarios para desarrollar aplicaciones utilizando el Data Access Builder son los siguientes:

1. Se crea o elige un proyecto (project) en el workbench de VAJava.

2. Se crea o elige un paquete (package) bajo el proyecto.

3.- Se "arranca" el Data Access Builder (DAB) en el proyecto.

4.- Se crea un "mapeo" (modelo con correspondencias) desde el catálogo del RDBMS y sentencias SQL.

5.- Se generan beans a partir de este mapeo.

6.- Se crea la interfaz de usuario (GUI) utilizando el editor de composición visual

7.- Se utilizan los beans generados desde el editor de composición visual

La tabla Departamento

Cada fila de la tabla departamento representa a un departamento (unidad organizacional) de la organización.

En la diapositiva se aprecia una de las herramientas ofrecidas por uno de los principales sistemas DBMS : el DB2UDB de IBM.

La herramienta es el Centro de Control (o Control Center) donde pueden ubicarse **instancias** del DBMS (en el ejemplo mostrado hay sólo una denominada "DB2"), una o más **bases de datos** (en el ejemplo mostrado se encuentran las bases de datos "SAMPLE" y "EMPRESA") y los **objetos de base de datos** de cada una de estas (tablas índices, vistas, etc.).

Precisamente DEPARTAMENTO es una tabla de la base de datos EMPRESA.

En la diapositiva pueden apreciarse los datos contenidos en esta tabla.

Invocando al Data Access Builder

Para "arrancar" el Data Access Builder se debe :

1. Arrancar el Workbench de Visual Age for Java

2. Seleccionar un paquete y elegir de su menú de contexto.
Para acceder al menú de contexto de un objeto, como un paquete en este caso, puede darse clic derecho sobre el mismo (tal como se realizó para obtener la diapositiva) o también puede usarse el menú "Selected" del Workbench.

3. Elegir la opción Tools

4. Elegir la opción Data Access

5. Elegir la opción Create Data Access Beans...

Al elegir esta opción se arranca el Data Access Builder, que puede usarse para crear un "mapping" a un esquema de base de datos.

Este esquema es representado por íconos en la ventana de DAB, que en un primer momento sólo presentará al ícono correspondiente al paquete.

Selección y correspondencia con Base de Datos

Estando en la ventana del Data Access Builder (indicada con el número 1 en la diapositiva) ...

Para crear un "mapping" con un esquema de BD se elige la opción "Map Schema" del menú de contexto del paquete (menú Selected de la barra de menú o clic derecho para el menú "pop-up").

Se hace clic en el botón "Next" (siguiente) para ir a la ventana de selección de base de datos y método de mapping (indicada con el número 2 en la diapositiva).

En esta ventana el DAB permite especificar la base de datos fuente, que en este caso es DB2 y la base de datos particular, que en el caso del ejemplo es la base de datos "EMPRESA".

En esta misma ventana el DAB permite elegir entre dos métodos para el mapping :

1. Seleccionando las tablas/vistas directamente de la base de datos elegida

2. Especificando la sentencia SQL

Para la primera forma no es necesario indicar la sentencia SQL.

La segunda forma se adecua cuando se requiere combinar datos de más de una tabla, entonces la sentencia SQL corresponderá a un join (SELECT con más de una tabla en su cláusula FROM).

Al seleccionar el primer método y hacer clic en el botón "Next" se accederá a la ventana de selección de tablas (indicada con el número 3 en la diapositiva) donde se podrá especificar la o las tablas a corresponder.

El DAB ofrece una ventana adicional para documentar o describir el mapping del esquema.

Cada uno de estos mappings o correspondencias se representará con un ícono colocado en una jerarquía (estructura de árbol).

Propiedades

En la ventana del Data Access Builder se apreciará el bean DEPARTAMENTO y sus atributos.

Puede apreciarse un ícono diferente para el atributo llave primaria del ejemplo mostrado.

Cabe mencionar que no en todos los casos las tablas presentan llaves primarias. Esto puede traer ciertos inconvenientes ya que el Data Access Builder no genera métodos de recuperación, eliminación y actualización para tablas sin llave.

Para estos casos el DAB ofrece una facilidad para especificar la llave :

En el menú de contexto del atributo correspondiente, se elige la opción "properties" (propiedades) y en esta ventana se especifica en la correspondiente opción ("Data Identifier").

Otra de las consideraciones importantes que hay que considerar antes de generar las clases a partir del mapping es el driver JDBC que se utilizará y la ubicación (URL) de la base de datos. Esto se logra con la ventana de propiedades del bean de la tabla.

En la diapositiva pueden apreciarse tanto el driver como el URL especificados.

Generación de beans y clases

Para generar los beans y clases que podrán ser usados para construir aplicaciones con los objetos de base de datos especificados, se selecciona el menú "File" y la opción "Save and Generate".

Los fuentes de Java son automáticamente importados y compilados en el paquete.

En el ejemplo mostrado se generaron 18 clases que encapsulan el acceso a la base de datos para aplicaciones que lo requieran.

En la diapositiva se han descrito los beans generados.
Adicionalmente a estos el DAB genera tres tipos de clases :

Clases BeanInfo
Clases Formulario (Forms)
Una clase aplicación

Las clases BeanInfo permiten integrar los beans a herramientas de desarrollo de aplicaciones, proveyendo información sobre las propiedades, métodos y eventos de cada clase.

Estas clases permiten a la herramienta saber qué puede hacerse con la correspondiente clase del bean.

El Data Access Builder genera una clase BeanInfo por cada una de los beans descritos en la diapositiva :
<clase>BeanInfo
<clase>ManagerBeanInfo
<clase>DataIdBeanInfo
<clase>DataIdManagerBeanInfo
<clase>DatastoreBeanInfo

donde <clase> es Departamento para el caso mostrado.

Las clases Form son componentes para GUI que pueden ser utilizadas al momento de desarrollar un applet o aplicación.
Por ejemplo:
<clase>Form es un formulario para desplegar los campos de una fila de la tabla, de manera que puede recuperarse, agregarse, actualizarse o eliminarse una fila a la vez. Será utilizado en un ejemplo posterior.

Programación Visual usando los beans generados

La ventaja de esta generación de beans y clases por parte del Data Access Builder es que puedan ser utilizados en el desarrollo de aplicaciones que accedan a los objetos de base de datos.

Como ejemplo se mostrará los pasos para desarrollar una aplicación de consulta y actualización a la tabla de Departamento.

Se hará uso del componente visual DepartamentoForm generado por el DAB así como de los demás beans y clases Info desde el Editor de Composición Visual.

Agregando beans a la paleta del Editor de Composición Visual

Una de las facilidades brindadas por el Editor de Composición Visual es que permite integrar beans "externos" a la paleta de beans para el desarrollo visual ("drag & drop").

Para esto debe accederse al menú Bean y elegir la opción "Modify Palette".

En esta ventana puede incorporarse beans a grupos existentes...

Incluso permite crear nuevos grupos de beans.

En el caso del ejemplo mostrado se hace clic en el botón "New Category" y se crea el grupo (categoría) EMPRESA.

Para incorporar beans a un grupo puede utilizarse la facilidad de búsqueda (clic en botón "Browse") y seleccionar una o varias clases desde la ventana "Choose a valid class".

Finalmente se incluyen las clases seleccionadas en el grupo haciendo clic al botón "Add to Category".

Uso de beans visuales

La aplicación mostrará una fila a la vez para que pueda realizarse una de las siguientes operaciones sobre la misma

Agregar una nueva fila
Eliminar una fila
Consultar una fila
Modificar algunos datos de una fila

Para esto se usará el componente visual "DepartamentoForm" generado por el Data Access Builder para desplegar los datos de una fila.

Asimismo se ofrecerán cuatro botones para cada una de las operaciones mencionadas.

Los botones que pueden utilizarse pueden arrastrarse de la categoría AWT (beans tipo button) o de la categoría swing. Para el caso mostrado se usa AWT.

Para cada uno de los botones, debe "customizarse" la propiedad de texto, cambiándolo a los apropiados, por ejemplo "Agregar", "Eliminar", "Consultar" y "Actualizar".

Del grupo donde fueron colocados los beans generados se "arrastra" el componente DepartamentoForm.

Uso de beans no visuales

La aplicación deberá interactuar con el objeto de base de datos : tabla Departamento, por lo tanto se requerirá hacer uso del bean Departamento, cuyos objetos precisamente representan filas de la tabla.

Este es un bean no visual que , al haberse agregado a un grupo (en este caso a la nueva categoría EMPRESA), su ícono puede ser también arrastrado desde la paleta.

Las acciones que se desea realice la aplicación son las correspondientes a los botones.

Se requieren conexiones tipo "Evento-a-Método" :

Botón	Evento	Método
Agregar	actionPerformed	add
Eliminar	actionPerformed	delete
Consultar	actionPerformed	retrieve
Modificar	actionPerformed	update

Otros beans no visuales que se requieren para el desarrollo de la aplicación son :

DepartamentoDatastore cuyo objeto representa la conexión a la base de datos y se requerirá precisamente para conectarse, desconectarse y comprometer o deshacer las transacciones sobre la BD.

MessageBox, bean para ser usado en el despliegue de mensajes de error.

El bean DepartamentoDatastore se arrastrará desde la paleta, en el grupo creado : EMPRESA.

El bean MessageBox se encuentra en la categoría "Enterprise Access".

Cuando uno "programa visualmente" no importará el orden en que sean colocadas las conexiones pues la aplicación funcionará de acuerdo a los eventos que en tiempo de ejecución vayan ocurriendo.

Es por este motivo que ha podido colocarse las conexiones evento-a-método de los botones a sus respectivas operaciones.

Definitivamente lo primero que tiene que ocurrir es que la aplicación (en este caso el applet) se cargue.

Entonces para establecer la conexión a la base de datos apenas se cargue el applet, bastará establecer una conexión evento-a-método entre el evento ***init()*** del applet y el método ***connect()*** de bean DepartamentoDatastore.

Para controlar una posible falla o imposibilidad de conexión con la base de datos, puede conectarse esta última conexión con el bean MessageBox. Para esto establecer una conexión evento-a-método entre el evento ***exceptionOcurred*** y el método ***showException*** del bean MessageBox.

Para indicar el texto del mensaje, debe "abrirse" esta conexión (doble clic o "Open" en el menú de contexto) y establecer el indicador "Pass event data" (ver diapositiva).

De igual forma que para la conexión, deberá indicarse la funcionalidad para la desconexión de la base de datos, que debería ocurrir al salir del applet.

Para esto sólo se requerirá una conexión evento-a-método desde el evento ***destroy*** del applet y el método ***disconnect*** del DepartamentoDatastore.

De la misma manera que pueden mostrarse el mensaje ante una falla o imposibilidad de conexión con la base de datos, debe mostrarse cualquier falla en las operaciones correspondientes a los cuatro botones de acción.

Para esto deben conectarse las conexiones que van de cada botón al bean Departamento con el bean MessageBox. Para esto establecer una conexión evento-a-método entre el evento ***exceptionOcurred*** y el método ***showException*** del bean MessageBox.

Para indicar el texto del mensaje, debe "abrirse" cada conexión (doble clic o "Open" en el menú de contexto) y establecer el indicador "Pass event data" (ver diapositiva).

OOP con Java

Laboratorios

Contenido

Laboratorio 1

Objetivos :

El ambiente de desarrollo
- El IDE
- (~~Integrated~~ Intelligent Development Environment de Visual Age for Java)

Elementos de programas
- Proyectos
- Paquetes
- Clases
  - Applets
- Métodos

Edición de elementos de programas

La ventana "Consola" (Console)

La ventana para pruebas pequeñas (Scrapbook)

Búsqueda y aprovechamiento de las clases provistas
- La clase Math

Tema :

1.- Cree un proyecto llamado "Mi1erProyecto".

2.- Dentro del mismo cree un paquete con nombre "paquete1".

3.- Cree un applet haciendo uso del SmartGuide

4.- Ejecútelo para que "corra" en el Applet Viewer.

5.- Modifique el applet de manera que el texto que muestre sea "Este es mi primer applet". Ejecútelo pero ahora desde un browser de Web.
Sugerencia : utilice la facilidad de exportación proporcionada por Visual Age for Java.

6.- Cree una clase que pueda ejecutarse y que muestre el mensaje "Mi 1er programa".

Sugerencia : La clase debe contener un método main.
Utilice el método println.

7.- Calcule las siguientes expresiones sin necesidad de crear una clase ejecutable.

Sugerencias :
Busque los métodos correspondientes en la clase *Math* del paquete *java.lang*.
Utilice el Scrapbook

7.1.- La raíz cuadrada de 1,999

7.2.- El ángulo cuya tangente es 0.75

7.3.- Muestre tres números aleatorios

Solución :

1.- Cree un proyecto llamado "Mi1erProyecto"

Haga clic en el correspondiente botón de la barra de herramientas del Workbench. También puede elegir el menú File, luego la opción Quick Start y elegir las opciones Basic, Create project.

En la ventana del SmartGuide elija el botón de selección "Create a new project named" y escriba "Mi1erProyecto".

Haga clic en el botón Finish y espere un instante.

2.- Dentro del mismo cree un paquete con nombre "paquete1"

Haga clic en el correspondiente botón de la barra de herramientas del Workbench. También puede elegir el menú File, luego la opción Quick Start y elegir las opciones Basic, Create package.

En la ventana del SmartGuide escriba Mi1erProyecto en el campo de texto correspondiente a Project.
En la misma ventana elija el botón de selección "Create a new package named" y escriba "paquete1".

Haga clic en el botón "Finish" y espere un instante.

3.-Cree un applet haciendo uso del SmartGuide

Haga clic en el botón de la barra de herramientas correspondiente a "Create Applet" También puede invocar a la ventana "Quick Start" del menú File y allí seleccionar Basic/Create Applet.

Escriba en el campo Project : Mi1erProyecto y en el campo Package paquete1.
Si estuvo posicionado en este proyecto/paquete, los nombres los encontrará escritos para usted.

Escriba Mi1erApplet en el campo Applet name.

Deshabilite la opción "Browse applet when finished".
Deshabilite la opción "Compose the class visually".

Haga clic en el botón "Finish" (También puede hacerlo sobre el botón "Next" para dar un vistazo a todas las opciones del SmartGuide, pero no modifique ninguna de manera que se genere un applet de ejemplo).

Observe que se han generado tres métodos para el applet, que es una clase (puede observar el correspondiente ícono que lo especifica).

4.- Ejecútelo para que "corra" en el Applet Viewer.

Selecciónelo haciendo clic sobre él y luego haga clic sobre el botón de la barra de herramientas correspondiente a "Run".
También puede elegir el menú "Selected" y luego la opción "Run".
Una variante de esta alternativa es hacer uso del menú de contexto del elemento de programa : Posicionándose sobre él (en este caso sobre el nombre del applet creado) se hace clic derecho y aparece el mismo menú correspondiente a "Selected", donde - para lo que se quiere ahora - se puede elegir "Run".

5.- Modifique el applet de manera que el texto que muestre sea "Este es mi primer applet". Ejecútelo pero ahora desde un browser de Web.

Haga clic en el nombre de la clase (Mi1erApplet) y tendrá visible el código de programa en el panel inferior.
Allí modifique la línea de código a :

```
Font font = new Font("Dialog", Font.BOLD, 24);
String str = "Este es mi primer applet";
int xPos = 5;
```

Guarde esta nueva versión del programa haciéndo Ctrl+S.

Si trata de ir a otro elemento de programa (por ejemplo a otro método) VAJava preguntará si desea guardar los cambios efectuados. Esto será equivalente a lo anterior (Ctrl+S).

Ejecútelo desde el Applet Viewer para comprobar la correcta modificación.

Expórtelo a un directorio de su disco, de manera que pueda ser cargado desde un Browser :

En el Paint(Graphics)

```
g.setFont(font);
g.setColor(Color.red);
g.drawString(str, xPos, 50);
```

Seleccione el elemento de la clase (Mi1erApplet).
Desde el menú Selected (o del menú de contexto) seleccione "Export..."

Una vez en la ventana del SmartGuide correspondiente, seleccione la opción "Directory" para indicar que se exportará a un directorio del ambiente operativo.
Haga clic en el botón "Next".
En la siguiente ventana, escriba la ruta de un directorio donde se exportará, pudiendo ser incluso el directorio raíz (como se puede apreciar en la diapositiva).

Seleccione la opción .class para indicar que se exportará el bytecode del programa Java, que luego podrá ser especificado en una página y ser cargado por un Browser que soporte Java.

Seleccione la opción .html para que se genere una página HTML con las instrucciones HTML apropiadas para cargar el applet.

Haga clic en el botón "Finish".

Desde el ambiente operativo ejecute un browser de Internet que acepte Java y cargue el archivo HTML generado, que en este caso tiene por nombre Mi1erApplet.html.

6.- Cree una clase que pueda ejecutarse y que muestre el mensaje "Mi 1er programa".

Posiciónese en el paquete "paquete1" de manera que la clase a crear esté contenida en éste.

Haga clic en el botón de la barra de herramientas correspondiente a "Create Class".
También puede invocar a la ventana "Quick Start" del menú File y allí seleccionar Basic/Create Class.
Otra alternativa para agregar elementos de programa es usar el menú "Selected" o menú de contexto y elegir la opción "Add" y luego elegir "Class...".

Todas las anteriores llevan a la ventana del SmartGuide.

Escriba en el campo Project : Mi1erProyecto y en el campo Package paquete1.
Si estuvo posicionado correctamente en este proyecto/paquete, los nombres los encontrará ya escritos para usted.

Escriba Mi1eraClase Ejecutable el campo "Class name".

Escriba java.class.Object el campo "Superclass"
LANG

Deshabilite la opción "Browse applet when finished".
Deshabilite la opción "Compose the class visually".

Haga clic en el botón "Next".

En la ventana Attributes del SmartGuide sólo deje habilitada la opción :
main(String[ ])
de manera que se genere este método para que la clase pueda ser ejecutada.

Haga clic en el botón "Finish".

Aprecie que una nueva clase con el nombre indicado aparece en la jerarquía del Workbench.

Para lograr que muestre un mensaje utilice el método println del objeto out, miembro estático de la clase System :

System.out.println("Mi 1er programa");

Haga Ctrl+S para guardar el programa.

Ejecútelo con el botón "Run"

Podrá apreciar la Consola del VAJava.

7.- Utilice el Scrapbook para calcular las expresiones

Primero busque la clase *Math* en el paquete java.lang del proyecto *Java Class libraries* (La biblioteca de clases de Java), donde podrá encontrar los métodos correspondientes.

Como el fuente de cada método está disponible, puede leer la documentación donde explica la funcionalidad de cada uno.

Para arrancar el Scrapbook, Del menú "Window" elija la opción "Scrapbook".

7.1.- La raíz cuadrada de 1.999
Utilice el método *sqrt*.

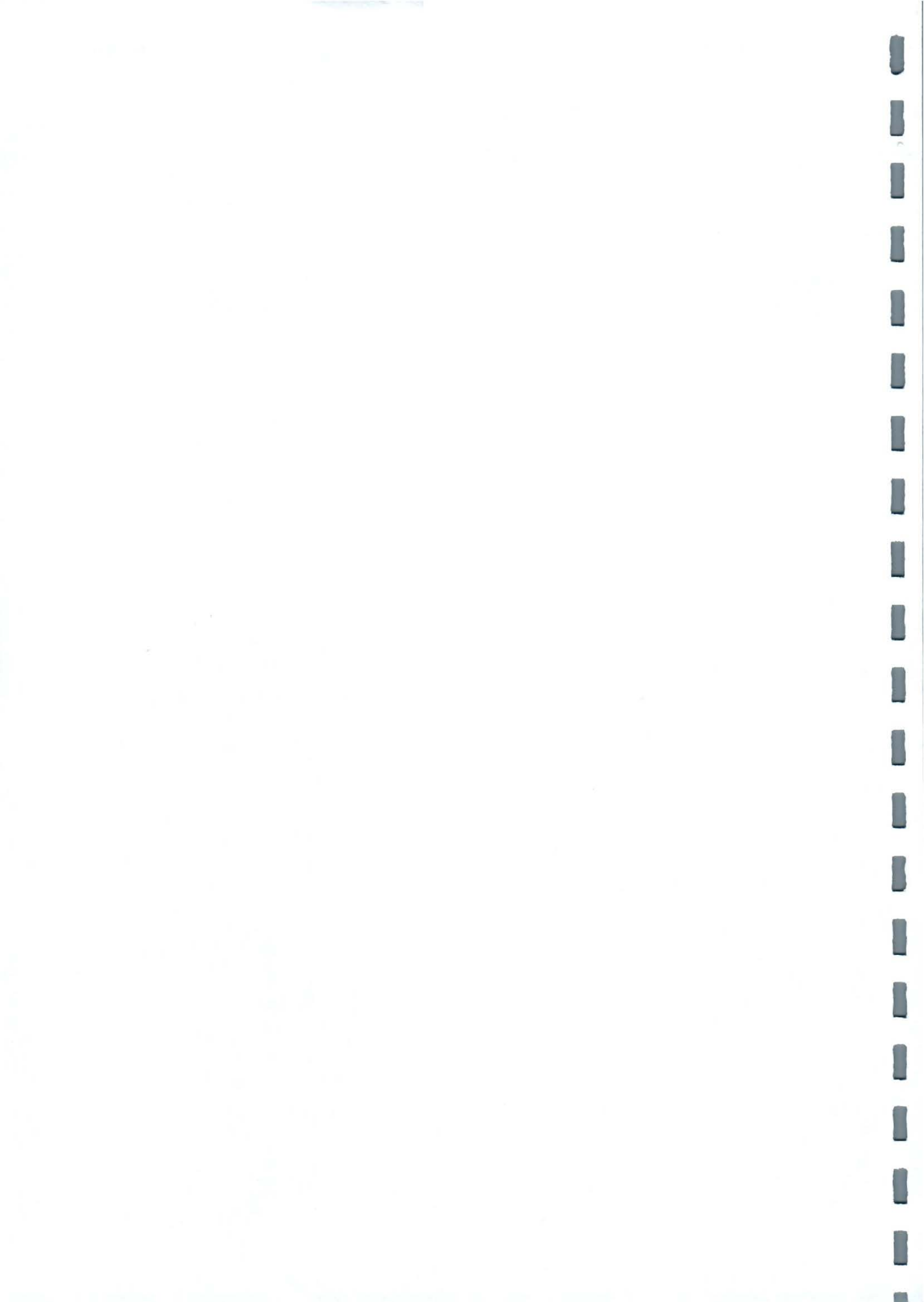
7.2.- El ángulo cuya tangente es 0.75
Utilice el método *atan*.

7.3.- Muestre tres números aleatorios

En la diapositiva puede apreciarse el documento fuente de HTML donde se especifican los tags necesarios para ejecutar el applet que ya se encuentra en los directorios especificados desde VAJava.

Utilice el método *random*.

Los resultados puede verificarlos en la consola.



Laboratorio 2

Objetivos :

Programas ejecutables
- método main
- Argumentos de un programa ejecutable

Operadores
- operadores enteros

Estructuras de control
- if
- while
- do - while

Tema 1 :

Desarrollar un programa que imprima todos los divisores de un número entero dado como argumento del programa.
Utilice la sentencia *while*.

Sugerencias :

Recuerde que los programas ejecutables tienen un método main que recibe argumentos tipo String (cadenas de caracteres).

Utilice el método estático *parseInt(String)* de la clase *Integer* para transformar el String leído como argumento en un entero y asignarlo a una variable entera para poder trabajarla.

Un divisor de un número es aquel que utilizado como divisor en una división, da como residuo cero.

Tema 2 :

Desarrollar un programa que calcule e imprima el máximo común divisor de dos enteros.
Utilice la estructura *do-while*.

Sugerencias :

Utilice el algoritmo de Euclides :

Sea i el mayor de los enteros y j el menor.

1.- Se calcula el residuo de la división entera entre i y j.

2.- Si el residuo fue cero entonces el máximo común divisor es j.

3.- Sino se reemplaza i por j y j por el residuo. Se va nuevamente al paso 1.

Solución del Tema 1 :

```
class Divisores {
public static void main(String args[]) {
     int numero  = Integer.parseInt(args[0]);
     int divisor = 1;
     System.out.println("Divisores de "+ numero);
     while (divisor <= numero) {
          if (numero%divisor == 0) {
               System.out.println(divisor);
          }
          divisor++;
     }
}
}
```

Solución del Tema 2:

```
class MaxComDiv {
public static void main(String[] args) {
     int i = Integer.parseInt(args[0]);
     int j = Integer.parseInt(args[1]);
     int residuo;
     System.out.print("Máximo comun divisor de "+
                              i +" y "+ j + " es ");
     if (i<j) {
          residuo = i; // uso temporal
          i = j;
          j = residuo;
     }
     do {
          residuo = i % j;
          i = j;
          j = residuo;
     }while (residuo != 0);

     System.out.println(i);
}
}
```

Laboratorio 3

Objetivos :

Programas ejecutables
- método main

Clases
- Método constructor
- Sobrecarga de métodos
- this
- Herencia
- Invocación de métodos

Tema :

1.- Desarrollar una clase FiguraGeometrica, de manera de poder derivar a partir de ella subclases de figuras geométricas específicas.
Añada una propiedad "nombre" para el nombre de la figura.
Añada un método imprimeNombre().

2.- Desarrollar una clase Cuadrado que contenga como dato el lado y como método area().

Implemente dos constructores, uno que asuma un valor de lado del Cuadrado por omisión y otro que acepte el valor.

3.- Desarrollar una clase Triangulo que contenga como dato los lados y como método perimetro().

Implemente dos constructores, uno que acepte los valores de los tres lados y otro que al recibir sólo dos valores, asuma que se trata de los "catetos" un triángulo rectángulo (los lados que forman el ángulo recto de 90º).
Sugerencia :
En el caso del triángulo rectángulo, debe calcular el tercer lado (la hipotenusa), para esto recuerde el teorema de Pitágoras : el valor de la hipotenusa el la igual raíz cuadrada de la suma de los cuadrados de los catetos.

4.- Desarrollar una clase Circulo que contenga el dato diámetro y como métodos area() y perimetro().
Sugerencia:
Utilice el valor de la constante PI provista por la clase *Math* del paquete *java.lang*.

Implemente un sólo constructor que exija el valor del diámetro.

5.- Desarrolle una clase ejecutable donde pueda comprobar todos los métodos invocados sobre los objetos correspondientes.

Compruebe que se puede invocar el método de la superclase a cualquier objeto de una subclase de ésta.

Solución :

```
abstract class FiguraGeometrica {
      String nombre;

      void imprimeNombre() {
            System.out.println(nombre);
      }
}


class Cuadrado extends FiguraGeometrica {
      float lado;

      Cuadrado() {
            this(1.0f);
      }

      Cuadrado(float lado) {
            this.nombre = "cuadrado";
            this.lado = lado;
      }

      float area() {
            return (lado*lado);
      }
}
```

```
class Triangulo extends FiguraGeometrica {
     float lado1, lado2, lado3;
     Triangulo(float lado1, float lado2) {
          this(lado1, lado2,
          (float)Math.sqrt((lado1*lado1)+(lado2*lado2)));
     }

     Triangulo(float lado1, float lado2, float lado3) {
          this.nombre = "triángulo";
          this.lado1 = lado1;
          this.lado2 = lado2;
          this.lado3 = lado3;
          if ( (lado1+lado2<lado3) ||
                (lado1+lado3<lado2) ||
                (lado2+lado3<lado1) )
               this.nombre = "triángulo incoherente";
     }

     float perimetro() {
          return lado1 + lado2 + lado3;
     }

}


class Circulo extends FiguraGeometrica {
     float diametro;
     Circulo(float diametro) {
          this.nombre = "círculo";
          this.diametro = diametro;
     }

     float area() {
          return (float)(Math.PI*diametro*diametro/4);
     }

     float perimetro() {
          return (float)(Math.PI*diametro);
     }

}
```

```
class CreaFiguras {
public static void main(java.lang.String[] args) {

     Cuadrado mi1erCuadrado = new Cuadrado();
     System.out.println("El area del "+ mi1erCuadrado.nombre
                         +" de lado "+ mi1erCuadrado.lado
                         +" es "+ mi1erCuadrado.area() );

     Cuadrado mi2doCuadrado = new Cuadrado(2.5f);
     System.out.println("El area del "+ mi2doCuadrado.nombre
                         +" de lado "+ mi2doCuadrado.lado
                         +" es "+ mi2doCuadrado.area() );

     Triangulo mi1erTriangulo = new Triangulo (2,3,4);
     System.out.println("El perímetro del "+
                          mi1erTriangulo.nombre
                         +" de lados "+
                          mi1erTriangulo.lado1
                         +", "+ mi1erTriangulo.lado2
                         +", "+ mi1erTriangulo.lado3
                         +" es "+
mi1erTriangulo.perimetro());

     Triangulo mi2doTriangulo = new Triangulo (3,4);
     System.out.println("El perímetro del "+
                          mi2doTriangulo.nombre
                         +" de lados "+
                          mi2doTriangulo.lado1
                         +", "+ mi2doTriangulo.lado2
                         +", "+ mi2doTriangulo.lado3
                         +" es "+
mi2doTriangulo.perimetro());

     Circulo mi1erCirculo = new Circulo(2.5f);
     System.out.println("El area del "+ mi1erCirculo.nombre
                         +" de diámetro "+
                          mi1erCirculo.diametro
                         +" es "+ mi1erCirculo.area() );

     // Invocando método de la superclase :

     mi1erCuadrado.imprimeNombre();
     mi1erTriangulo.imprimeNombre();
     mi1erCirculo.imprimeNombre();
}

}
```

Laboratorio 4

Objetivos :

Sentencias condicionales
- if
- switch

Sentencias para Bucles (loops)
- for

Tratamiento de cadenas de caracteres.

Tema :

Elaborar un programa que permita contar la cantidad de caracteres, palabras (incluye números, puntuaciones, etc., esto es lo que se denomina "tokens" en la jerga de compiladores y analizadores lexicales) y líneas de un texto incluido como una cadena de texto dentro del programa.

Utilizar el siguiente texto para verificación con el solucionario :

```
"Las organizaciones que para el año 2000
no consideren entre sus objetivos brindar a
sus usuarios finales acceso autónomo a los datos,
arriesgan su posición competitiva"
```

Al ejecutar el programa debe obtenerse es siguiente resultado :

```
caracteres = 171
tokens     = 26
lineas     = 4
```

Sugerencias :

Crear una variable "texto" de tipo String para incorporar el texto ejemplo.

Incluirle al texto los caracteres "\n" para los saltos de línea (son 4 líneas).

Utilizar el método ***length()*** para obtener la longitud (en cantidad de caracteres de la cadena)

"Barrer" la cadena con un bucle que avance de caracter en carácter y utilice el método ***charAt(i)*** para recuperar el carácter de la posición *i* de la cadena de caracteres. Haga que el bucle vaya desde el primer carácter hasta el último, cuyo número se obtuvo con length().

Para contar caracteres (a pesar que ya obtuvo la respuesta con el método length()), utilice un contador en el bucle de barrido.

Para contar líneas utilice un contador que se incremente cada vez que se encuentre con un carácter de cambio de línea ("\n")

Para contar palabras utilice un contador que sólo sea incrementado en caso que encuentre un caracter invisible (espacio (" "), tabulador ("\t") o cambio de línea ("\n")) pero el anterior caracter no haya sido uno de estos mismos.

Solución :

```
public class CuentaTokensDeString {
      static String texto = "Las organizaciones que para el año 2000 \n"+
                  "no consideren entre sus objetivos brindar a \n"+
                  "sus usuarios finales acceso autónomo a los datos, \n"+
                  "arriesgan su posición competitiva\n";
      static int longitud = texto.length();

public static void main(java.lang.String[] args) {
      boolean barriendoToken = false;
      int     caracteres     = 0;
      int     tokens         = 0;
      int     lineas         = 0;
      for (int i = 0 ; i < longitud; i++) {
            char c = texto.charAt(i);
            caracteres++;
            switch (c) {
                  case '\n':      //cambio de línea
                        lineas++;
                  case '\t':      // o tabulador

                  case ' ' :      // o espacio en blanco
                        if (barriendoToken) {
                              tokens++;
                              barriendoToken = false;
                        }
                        break;
                  default :       // cualquier otro caracter
                        barriendoToken = true;
            }
      }
      System.out.println(     "caracteres = " +caracteres+ "\n"+
                              "tokens     = " +tokens+ "\n"+
                              "lineas     = " +lineas);
}

}
```

Introducción al Laboratorio 5

La interfaz **MouseListener**

Es subclase de EventListener y sirve para recibir eventos del mouse sobre un componente.

Métodos que deben ser escritos por objetos que implementen esta interfaz :

mouseClicked(MouseEvent e)
Invocado cuando se hace clic (con el mouse) sobre un componente.

mouseEntered(MouseEvent e)
Invocado cuando el puntero del mouse "entra" a un componente.

mouseExited(MouseEvent e)
Invocado cuando el puntero del mouse "sale" de un componente.

mousePressed(MouseEvent e)
Invocado cuando se presiona uno de los botones del mouse sobre un componente

mouseReleased(MouseEvent e)
Invocado cuando se suelta uno de los botones del mouse sobre un componente

La interfaz **MouseMotionListener**

Es subclase de EventListener y sirve para recibir eventos de movimiento del mouse sobre componentes visuales.

Métodos que deben ser escritos por objetos que implementen esta interfaz :

mouseDragged(MouseEvent e)

Es invocado cuando uno de los botones del mouse es presionado sobre un componente para luego arrastrar el dispositivo (drag). El evento continúa mientras no se suelte el botón del mouse.

mouseMoved(MouseEvent e)

Invocado cuando el mouse ha sido movido sobre un componente.

Como puede apreciarse, todos los métodos tienen como parámetro al objeto evento, en este caso evento del mouse (MouseEvent).

El objeto evento de mouse tiene dos métodos importantes y que serán requeridos para el desarrollo del tema del laboratorio :

getX() retorna la posición horizontal (x) relativa al componente fuente.
getY() retorna la posición vertical (y) relativa al componente fuente.

Laboratorio 5

Objetivos :

Programación con Eventos
    Eventos del mouse

Applets
    clase Container

Gráficos
    clase Graphics
    clase Rectangle

Tema :

Desarrollar un applet que permita dibujar rectángulos con el mouse: cada rectángulo debe dibujarse desde el punto donde se presione el botón del mouse hasta donde se suelte luego de "arrastrarlo". Este último punto será la esquina contraria del rectángulo que se graficará.

La forma de cada rectángulo debe visualizarse mientras que se vaya "arrastrando" (drag) el mouse y quedar permanente sólo cuando se suelte el botón del mismo.

Sugerencias :

El applet debe implementar las interfaces MouseListener y MouseMotionListener para aceptar eventos del mouse.

Establecer, con una CONSTANTE un límite para la cantidad de rectángulos a dibujar.

Utilice un arreglo de objetos tipo Rectangle, de manera que pueda dibujar a todos a la vez cuando se requiera (por ejemplo pintar el applet).

En la inicialización del applet, registrarlo para recibir eventos originados sobre él mismo :
Para esto utilizar los métodos addMouseListener y addMouseMotionListener.
addMouseListener(MouseListener) registra al objeto MouseListener como apto para recibir eventos del componente donde es invocado el método.
addMouseMotionListener(MouseMotionListener) registra al objeto MouseListener como apto para recibir eventos del componente donde es invocado el método.

Para controlar que los rectángulos sean dibujados dentro del área del applet haga uso de los métodos MouseEntered y MouseExited. Puede definir una variable lógica y aprovecharla para no dibujar cuando se esté fuera de la ventana.

Establezca el punto inicial del rectángulo cuando se presione el botón del mouse. Aproveche este momento para también inicializar (a cero) el ancho y alto del rectángulo a dibujar.

Cree definitivamente el objeto rectángulo cuando el botón del mouse sea soltado.

Para evitar hacer la aplicación más compleja, concéntrese sólo en valores positivos de avance (horizontal y vertical) del mouse sobre el área del applet.

Para lograr el efecto de ir visualizando el rectángulo mientras que se vaya arrastrando el mouse (moviéndolo sin ser soltado), debe dibujarlo en cada instante del arrastre pero borrando su imagen anterior. Para borrarlo puede dibujarlo (con las medidas anteriores del arrastre) con el color de fondo y luego dibujarlo con las nuevas medidas pero con el color de dibujo.

Se recomienda usar dos colores que hagan fuerte contraste. Por ejemplo blanco para el fondo (background) y negro para el dibujo de los rectángulos.

Solución :

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class DibujaRectangulos extends Applet implements MouseListener,
MouseMotionListener {

  private  int    xInicio, yInicio,  // Punto inicial del clic
            ultimoAncho,         // Ultimo ancho dibujado mientras arrastre
            ultimoAlto;          // Ultimos alto dibujado mientras arrastre
  final    int MAXFIGURAS = 9;     // Máxima cantidad de figuras a dibujar
  private  int dibujados  = 0;     // Cantidad de rectángulos dibujados
  private  boolean dentro = true; // Indicador de estar dentro del área

  private  Color  color      = Color.black; // Color para los rectángulos

  private  Rectangle[] rectangulo = new Rectangle[MAXFIGURAS];


public void init() {
      super.init();
      addMouseListener(this);
      addMouseMotionListener(this);
      setBackground (Color.white);
}


public void mousePressed(MouseEvent e) {
      if (dibujados < MAXFIGURAS) {
            xInicio     = e.getX();
            yInicio     = e.getY();
            ultimoAncho = ultimoAlto = 0;
      }
}


public void mouseDragged(MouseEvent e) {
      int x = e.getX();
      int y = e.getY();
      int ancho = x - xInicio;
      int alto  = y - yInicio;

      if ((dibujados < MAXFIGURAS) &&  (ancho > 0) && (alto > 0))
      { Graphics g = getGraphics();
        g.setColor (getBackground ());
        g.drawRect (xInicio, yInicio, ultimoAncho, ultimoAlto);
        g.setColor (color);
        g.drawRect (xInicio, yInicio, ancho, alto);
        ultimoAncho = ancho;
        ultimoAlto  = alto;
      }
}
```

```
public void mouseReleased(MouseEvent e)
{   int x = e.getX();
      int y = e.getY();
      int ancho = x - xInicio;
      int alto  = y - yInicio;
      if ((dibujados < MAXFIGURAS) && (ancho > 0) && (alto > 0)){
            rectangulo[dibujados++] = new Rectangle (xInicio, yInicio,
ancho, alto);
      }
      repaint();
}

public void mouseEntered(MouseEvent e) {
      dentro = true;
      repaint();
}

public void mouseExited(MouseEvent e) {
      dentro = false;
      repaint();
}


public void paint(Graphics g) {
      if (dentro == true) {
            for (int i = 0; i < dibujados; i++) {
                  g.drawRect(rectangulo[i].x,
                             rectangulo[i].y,
                             rectangulo[i].width,
                             rectangulo[i].height);
            }
      }else {
            g.drawString ("No salga de la ventana...", 10, 50);
      }
}

public void mouseMoved(MouseEvent e) {}


public void mouseClicked(MouseEvent e) {}

}
```

Introducción al laboratorio 6

El anglicismo "layout" se refiere a la distribución física de objetos en una determinada zona.
En el caso de GUIs, los objetos son componentes gráficos y la zona es el objeto contenedor (de tipo *Container*) que los contiene.
El paquete awt provee un conjunto de clases manejadoras de layout (Layout managers o que implementan la interfaz *LayoutManager*) que permiten indicar la manera como los componentes deben ser ubicados y/o ajustados al momento de ser mostrados.

Para establecer un layout a un contenedor (objeto de clase *Container*) se debe invocar el método *setLayout*(*LayoutManager*). Cuando ya se encuentra establecido el layout al contenedor, cualquier componente que sea agregado a éste, también será agregado al layout del contenedor.

La clase **FlowLayout**
Coloca los componentes en fila mientras quepan en el contenedor, en este último caso los coloca en la siguiente fila. El orden es el de agregado al contenedor. Este layout es el válido por omisión para la clase *Pane* y por lo tanto para *Applet*s.
Provee métodos para especificar espaciamiento entre componentes y tamaños.

La clase **GridLayout**
Coloca los componentes en filas y columnas según el método constructor elegido y los parámetros especificados.

La clase **BorderLayout**
Coloca y ajusta el tamaño de los componentes de manera que llenen completamente el contenedor. Este layout es el válido por omisión para la clase *Window*, por lo tanto *Frame*s y ventanas de *Dialog*o la usarán si no se especifica otro layout.

La clase **CardLayout**
Coloca los componentes, sin ajustar sus tamaños, al estilo de páginas una detrás de otra (sólo se ve un componente a la vez) y provee los métodos first(), next(), previous() y last() para moverse hacia los componentes.

La clase **GridBagLayout**
Es la más flexible pues presenta más opciones. La base es un grid (aunque no es una extensión de *GridLayout*) pero los componentes no están restringidos a celdas e incluso pueden ocupar más de una. Las características y restricciones para cada componente que se agregará a un contenedor con este layout son especificadas con un objeto de tipo *GridBagConstraints*, que se agrega al layout junto con el componente.

Laboratorio 6

Objetivos :

Applets

Interfaz Gráfica al Usuario (GUI).

Distribución de componentes en la interfaz.

Tema 1 :

Elabore un applet que presente 5 botones dispuestos en fila

Tema 2 :

Elabore un applet que presente 5 botones de la siguiente manera :
Uno en la parte superior, otro en la inferior, otro a la izquierda y otro a la derecha.
Por último debe incluirse un botón al centro.
Todos los componentes deben "llenar" completamente el área del applet.

Tema 3 :

Elabore un applet que presente un botón, una etiqueta y un panel que contenga un área de texto. Los tres componentes deben estar traslapados en el applet y deben poder visualizarse uno a la vez.
Para pasar a ver el siguiente componente debe hacerse clic al mouse.

Sugerencias :

El applet debe aceptar eventos del mouse, por lo que debe implementar la correspondiente interfaz.

Cada componente debe registrar al applet para enviarle el clic del mouse.

No olvidar que todos los métodos de la interfaz deben ser implementados, aunque no sean utilizados.

Solución al Tema 1 :

```
import java.applet.*;

import java.awt.*;

public class BotonesEnFila extends Applet {

     public void init() {
          setLayout(new FlowLayout(FlowLayout.RIGHT));
          add(new Button("1er botón"));
          add(new Button("2do botón"));
          add(new Button("3er botón"));
          add(new Button("4to botón"));
          add(new Button("5to botón"));
     }

}
```

Solución al Tema 2 :

```
import java.applet.*;

import java.awt.*;

public class BotonesAlBorde extends Applet {

     public void init() {
          setLayout(new BorderLayout());
          add(new Button("Superior"),"North");
          add(new Button("Inferior"),"South");
          add(new Button("Izquierda"),"West");
          add(new Button("Derecha"),"East");
          add(new Button("Centro"));
     }
}
```

Solución al Tema 3 :

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ComponentesTraslapados extends Applet implements MouseListener
{
      Button   boton1;
      Label    label2;
      TextArea texto3;
      CardLayout layout;

      public void init() {
            setLayout(layout = new CardLayout());
            add(boton1 = new Button("botón1"), "boton1");
            boton1.addMouseListener(this);

            add(label2 = new Label("Página con texto..."), "label1");
            label2.addMouseListener(this);

            Panel panel = new Panel();
            panel.setLayout(new BorderLayout());

            panel.add(texto3 = new TextArea("texto en página panel"));
            texto3.addMouseListener(this);
            add(panel,"panel");
      }

      public void mouseClicked(MouseEvent e) {
            layout.next(this);
      }

      public void mouseEntered(MouseEvent e){}
      public void mouseExited(MouseEvent e){}
      public void mousePressed(MouseEvent e){}
      public void mouseReleased(MouseEvent e) {}
}
```

Introducción al Laboratorio 7

Tanto las aplicaciones que tienen que interactuar con el usuario como aquellas que no, tienen que manejar (recibir y entregar) datos de alguna manera. Las primeras lo hacen desde y hacia dispositivos manipulados por el usuario como un teclado y un monitor, mientras las segundas lo hacen con dispositivos de almacenamiento y comunicación de datos de datos como un disco magnético o un conector (socket) de red.

Java llama "flujo" (stream) a la abstracción correspondiente a la entrada/salida de datos desde y hacia dispositivos, de manera que se encapsula la complejidad de diferenciar los diferentes tipos de dispositivos (el código de programa no debe preocuparse de diferenciar entre un teclado y un socket de red).

La entrada (desde cualquier dispositivo) está encapsulada en la clase ***InputStream*** y la salida en la clase ***OutputStream***. Java también provee subclases de estas precisamente para manejar las diferencias entre dispositivos (archivos de disco, socket y hasta buffers de memoria).

Las operaciones de E/S son tan comunes, que el correcto manejo de las excepciones que puedan originarse es un aspecto fundamental para el éxito en el desarrollo de aplicaciones de calidad. Java provee un paquete completo, ***java.io***, para este tratamiento.

La clase File

Su función principal es hacer referencia a un archivo de disco.
No especifica la forma de almacenamiento ni recuperación de la información en los archivos, sino más bien describe las propiedades de objetos de tipo archivo.

La mayoría de sistemas operativos siguen tratando a los directorios como jerarquías de archivos. Por este motivo Java también incluye a los directorios en esta clase.

Laboratorio 7

Objetivos :

Métodos de la clase File

Manejo de Excepciones
Excepción IOException

Obtención de datos a partir de archivos en disco.
Clases :
InputStream
OutputStream
FileDescriptor
FileInputStream
FileOutputStream
BufferedInputStream
BufferedOutputStream

Tema 1 :

Elaborar un programa que muestre las siguientes características de un archivo en disco :

Nombre del archivo
Ruta de acceso
Ruta absoluta de acceso
Indicador si puede leerse de él
Indicador si puede escribirse en él
Indicador si es un directorio
Fecha de última modificación
Tamaño del archivo en bytes

Para probar su programa deberá crear un directorio en la raiz del disco (por ejemplo puede llamarlo “dir”) y un subdirectorio en este (por ejemplo llamado “subdir”).
En este subdirectorio cree un archivo simple, por ejemplo de tipo texto con extensión .TXT.
Use un editor (por ejemplo notepad) para editar un poco de caracteres y así el archivo obtenga cierto tamaño y pueda medirse éste en bytes.

Tema 2 :

Elaborar un programa que muestre el contenido de un directorio especificado dentro del programa.
En la salida deberá poder diferenciarse entre directorios y archivos.

Para probar su programa deberá crear un directorio en la raiz del disco (por ejemplo puede llamarlo "dir") y dos subdirectorios en este (por ejemplo llamados "subdir1" y "subdir2").
En este subdirectorio cree también un archivo simple, por ejemplo de tipo texto con extensión .TXT, por ejemplo "archivo2".

Sugerencias :

Primero consistencie si el objeto File es un directorio; utilice el método *isDirectory()* .

Capture el resultado del método *list()* (que retorna el contenido de un directorio) en un arreglo de Strings.

Mediante un bucle avance en el arreglo obtenido e imprima los contenidos con un texto diferente para directorios y para archivos.

Tema 3 :

Elaborar un programa que permita contar la cantidad de caracteres, palabras (incluye números, puntuaciones y "tokens" en general) y líneas de un texto incluido en uno o varios archivos en disco, especificados como parámetros al momento de ejecutar la aplicación.
En caso que no se indiquen archivos a leer, el programa debe permitir ingresar el texto desde teclado.

Incluir el siguiente texto en un archivo "`c:\dir\subdir1\archivo.txt`" para verificación con el solucionario :

```
"Las organizaciones que para el año 2000
no consideren entre sus objetivos brindar a
sus usuarios finales acceso autónomo a los datos,
arriesgan su posición competitiva"
```

Al ejecutar el programa con el argumento correspondiente al archivo, debe obtenerse es siguiente resultado :

```
caracteres = 172
tokens     = 26
lineas     = 4
```

Solución al Tema 1 :

```
import java.io.File;

class ArchivoPropiedades {

public static void main(String[] args) {
      //File a = new File(args[1]);
      File a = new File("c:/dir/subdir/archivo.txt");
      System.out.println("Nombre del archivo : "+a.getName());
      System.out.println("Ruta de acceso : "+a.getPath());
      System.out.println("Ruta absoluta de acceso : "+a.getAbsolutePath());
      System.out.println("Indicador si puede leerse de él : "+a.canRead());
      System.out.println("Indicador si puede escribirse en él :
"+a.canWrite());
      System.out.println("Indicador si es un directorio :
"+a.isDirectory());
      System.out.println("Fecha de última modificación :
"+a.lastModified());
      System.out.println("Tamaño del archivo en bytes : "+a.length());
}
}
```

Solución al Tema 2 :

```
import java.io.File;

class Directorio {

public static void main(java.lang.String[] args) {
        String raiz = "c:/dir";
        File d = new File(raiz);
        if (d.isDirectory()) {
                String s[] = d.list();
                for (int i=0;i<s.length;i++) {
                        File f = new File(raiz + "/" + s[i]);
                        if (f.isDirectory())
                                System.out.print("Directorio : ");
                        else
                                System.out.print("Archivo : ");
                        System.out.println(s[i]);
                }
        }else {
                System.out.println(raiz + " no es un directorio");
        }
}

}
```

Solución al Tema 3 :

```
import java.io.*;

class CuentaTokensDeArchivo {
     static int caracteres = 0;
     static int tokens     = 0;
     static int lineas     = 0;

public static void cuenta(InputStream f) throws IOException {
     int c = 0;
     boolean ultimoNoInvisible = false;
     String invisibles = " \t\n\r";
     while ( (c = f.read() ) != -1) {
          caracteres++;
          if (c == '\n') {
               lineas++;
          }
          if (invisibles.indexOf(c) != -1) {
               if (ultimoNoInvisible) {
                    tokens++;
               }
               ultimoNoInvisible = false;
          }else {
               ultimoNoInvisible = true;
          }
     }
}
```

```
public static void main(String[] args) {
     BufferedInputStream f;
     try {
          if (args.length == 0) {
               FileDescriptor fd = new FileDescriptor();
               f = new BufferedInputStream (new
                                   FileInputStream(fd.in));
               cuenta(f);
          }else {
               for (int i=0; i<args.length;i++) {
                    File archivo = new File (args[i]);
                    if (archivo.exists() && archivo.isFile()
                              && ! archivo.isDirectory()) {
                         f = new BufferedInputStream (new
                                   FileInputStream(archivo));
                         cuenta(f);
                    }else {
                         System.out.println(args[i]+" no es
                                             archivo válido");
                    }
               }
          }
     }catch (IOException e) {
          System.out.println(" "+e);
          return;
     }
     System.out.println( "caracteres = " +caracteres+ "\n"+
                         "tokens     = " +tokens+ "\n"+
                         "lineas     = " +lineas);
}

}
```

Introducción al laboratorio 8

Importante.- El siguiente laboratorio requiere los conocimientos de hilos y sincronización en Java (haber escuchado la correspondiente sesión teórica).

La sincronización entre procesos puede lograrse usando el modificador synchronized en los métodos adecuados o también bloqueando todo un bloque de código de programa mediante la sentencia *synchronized*.

Esto permite descartar la programación basada en bucles (loops) de suceso y dividir las tareas en hilos.

El "sondeo" también es una técnica tradicional mediante la cual se comprueba una y otra vez si se cumple una determinada condición, para en ese instante realizar alguna acción.

Por ejemplo el tema propuesto es la implementación de una cola, donde una clase produce objetos (números enteros) para dejarlos en una cola y otra los retira o "consume" de la misma cola. Si el requerimiento es tal que se necesite que ambas acciones estén sincronizadas (el productor debe esperar que se haya consumido un objeto antes de colocar otro en la cola), podría adoptarse una estrategia de programación por sondeo utilizando un bucle infinito mientras el otro proceso no haya terminado (usando una variable lógica indicadora de este evento). El problema de este tipo de solución es el desperdicio de recursos (CPU) en el bucle.

Java proporciona mecanismos de comunicación entre procesos a través de los métodos *wait* , *notify* y *notifyAll*, que son métodos implementados como *final* en la clase *Object* (por lo tanto todo objeto Java dispone de ellos). Cualquiera de estos métodos puede ser llamado desde dentro de un método *synchronized*.

wait

Le indica al hilo en curso que abandone el monitor y que "duerma" hasta que otro hilo entre en el mismo monitor y llame a *notify*.

notify

"Despierta" al primer hilo que realizó una llamada a *wait* sobre el mismo objeto.

notifyAll

Despierta a todos los hilos que realizaron una llamada a *wait* sobre el mismo objeto. El hilo despertado que tiene la mayor prioridad será el primero en ejecutarse.

Laboratorio 8

Objetivos :

Hilos
- clase Thread
- interfaz Runnable

Sincronización
- modifcador syncronized

Comunicación entre hilos
- método wait()
- método notify()

Tema :

Elabore una aplicación que introduzca y retire objetos de una cola de una manera sincronizada y sin desperdicio de recursos del procesador.

Sugerencias :

Desarrolle una clase que produzca objetos (no se complique y que sean simples números enteros) y los deje en otro objeto que sea la cola.

Cada vez que deje un objeto en la cola, imprima el suceso.

Desarrolle otra clase que retire los objetos (los "consuma") de la misma cola.

Cada vez que consuma un objeto de la cola, imprima el suceso.

El requerimiento es que se necesita que ambas acciones estén sincronizadas, entonces el productor debe esperar que se haya consumido un objeto antes de colocar otro en la cola y el consumidor debe esperar a que el productor haya dejado un objeto en cola para recién consumirlo. Implemente esto haciendo que el productor y consumidor sean hilos (o utilicen hilos implementando Runnable) y usando una variable lógica que no permita a estos realizar más de una acción a la vez (si lo intenta duérmalo !).

Solución :

```
class Cola {
      int n;
      boolean indicaPut = false;

      synchronized int get() {

            // while (!indicaPut) {}
            // hubiera sido un desperdicio de CPU !

            if (!indicaPut)
                  try {wait();}
                  catch(InterruptedException e){};
            System.out.println("Se obtuvo : "+ n);
            indicaPut = false;
            notify();
            return n;
      }

      synchronized void put(int n) {

            // while (indicaPut) {}
            // hubiera sido un desperdicio de CPU !

            if (indicaPut) {
                  try {wait();}
                  catch(InterruptedException e){};
            }
            this.n = n;
            indicaPut = true;
            System.out.println("Se colocó : "+ n);
            notify();
      }
}
```

```
class Consumidor implements Runnable {
     Cola cola;

     Consumidor(Cola cola) {
          this.cola = cola;
          new Thread(this, "Consumidor").start();
     }

     public void run() {
          int i = 0;
          while (true) {cola.get();}
     }
}


class Productor implements Runnable {
     Cola cola;
     Productor(Cola cola) {
          this.cola = cola;
          new Thread(this,"Productor").start();
     }
     public void run() {
          int i = 0;
          while (true) {cola.put(i++);}
     }
}


class ProductorYConsumidorUsanCola {

     public static void main(String args[]) {
          Cola cola = new Cola();
          new Productor(cola);
          new Consumidor(cola);
     }
}
```

Laboratorio 9

Objetivos :

Beans

El Editor de Composición Visual

Programación Visual
Conexiones

Tema :

Desarrolle un applet que presente un campo de texto y una lista de texto. Los items de texto que el usuario escriba en el campo de texto podrán pasarse a la lista mediante un clic de un botón con una etiqueta que indique esta función. Ítems de la lista de texto podrán eliminarse mediante clic en otro botón que indique esta función.

Sugerencias :

Cree el applet de manera visual

Utilice los beans de la categoría "swing" en la paleta de componentes.

Primero concéntrese en colocar los componentes requeridos (beans visuales : campo de texto, lista, botones, etiquetas) para luego modificar sus propiedades y ubicaciones en el área del applet.

Recuerde que en las clases de la biblioteca "swing", los datos y las vistas de los datos están en objetos separados, por lo que se requerirá un objeto (bean) para guardar los items que irán en el objeto visible (Lista). El bean que necesitará usar será el *DefaultListModel* que será un componente no GUI pero que lo verá iconizado para poder conectarlos con otros componentes.

Luego de incorporar este bean no visual, implemente la funcionalidad requerida mediante conexiones evento-método.

Para completar estas conexiones con los parámetros a pasar en los métodos, utilice conexiones parámetro-de-propiedad.

Solución :

Arranque VisualAge for Java

Una vez en el "Workbench" :

Del menú "File" elija "Quick Start"
Seleccione "Basic", luego "Create applet" y haga clic en "OK"

Llene la siguiente información en los campos de la ventana:
Project : Proyecto Laboratorio
Package : P_ListaItems
Applet name : ListaItems
Superclass : com.sun.java.swing.JApplet
Para esto último puede usar el botón "Browse" y elegir "JApplet" de la lista "Type Names"

Seleccione "Compose the class visually".
Haga clic en el botón "Finish"

Una vez en el editor de composición visual (VCE):

- Coloque un campo de texto

De la paleta de beans seleccione el bean "JTextField".

Observe que en la parte inferior el VCE le va informando la selección del momento.

Con el puntero en forma de cruz haca clic dentro del bean inicial que tiene forma de rectángulo discontinuo (JApplet) para dejar el bean seleccionado contenido en él.

El puntero vuelve a tomar forma de flecha y con el mouse puede arrastrar el campo de texto a cualquier otro lugar.

- Etiquételo con "Item"

Seleccione el bean "JLabel" y suéltelo en la parte superior del campo de texto.

Para cambiar el contenido del texto de la etiqueta tiene varias alternativas :

a) Hacer doble clic sobre el objeto
b) Seleccionar el menú "Tools" y luego la opción "Properties"
c) Hacer clic en el botón correspondiente de la barra de herramientas.

Una vez que aparece la pequeña ventana de propiedades, seleccione "text" y escriba "Item".
Presione "Enter" (vea que la etiqueta dice ahora "Item").

- Agregue un campo Lista y etiquételo con "Lista de Items" (debe ser "scroleable")

Seleccione el bean "JScrollpane" y colóquelo debajo del campo de texto.

Siga el mismo procedimiento anterior con el bean "JList", pero déjelo dentro del JScrollpane.
El tamaño de la lista se ajustará automáticamente con el JScrollpane.

Para colocar la etiqueta a la lista, puede seguir el procedimiento ya explicado. Otra alternativa es hacer uso de las facilidades de Copiar/Pegar :

Seleccione la etiqueta existente haciendo clic con el mouse y use el menú Edit/Copy para luego usar Edit/Paste y haga clic en la parte superior de la lista.
También puede usar las facilidades de Drag&Drop haciendo Ctrl+Clic y arrastrando el objeto copiado hasta la posición de la copia.

Cambie el texto de la etiqueta de la misma forma que el caso anterior. Probablemente deba agrandar el tamaño de esta etiqueta, lo que puede realizar con el mouse.

- Incluya los 2 botones

Siga el procedimiento anterior para incluir los botones de Agregar y Eliminar.

Para seleccionar más de una vez un bean, al momento de hacer clic en la paleta, presione al mismo tiempo la tecla "Ctrl".

Para deshabilitar esta facilidad (una vez haya colocado ambos botones) seleccione la herramienta de selección (flecha en la paleta).

Usando la ventana de propiedades cambie de nombre a los botones de "JButton1" a "Agregar" y de "JButton2" a "Eliminar".

Guarde el bean eligiendo la opción "Save bean" del menú "Bean".

Para acomodar los beans de manera alineada seleccione la herramienta lista de beans:

a) Use el menú Tools, opción "Bean List" o

b) El correspondiente botón de la barra de herramientas.

De la lista seleccione el JScrollpane y con el mouse haga Ctrl+Clic en el campo de texto.
Ajuste el ancho de ambos objetos, igualándolos, usando el menú "Tools"/"Match Width" o el botón correspondiente en la barra de herramientas.

Guarde el bean eligiendo la opción "Save bean" del menú "Bean".

Se requiere un bean para guardar los items que irán en el objeto visible JList. Para esto seleccione la herramienta de selección de beans (Choose Bean) de la paleta y luego para el tipo de bean "Class" busque (Browse) el bean DefaultListModel, haga clic en "OK" y luego colóquelo debajo del bean applet (el panel gris).

- Conecte el evento actionPerfomed del botón "Agregar" con el método "addElement" del bean DefaultListModel

Haga clic derecho sobre el botón "Agregar" , luego "Connect" y elija "actionPerformed"
Luego haga clic sobre el bean DefaultListModel.
Seleccione "Connectable Features". Haga clic en "addElement(java.lang.Object)". El hecho que la conexión aparezca discontinua, indicará que todavía no está completamente especificada. El motivo es que falta el parámetro del método addElement(java.lang.Object).

- Conecte el evento actionPerfomed del botón "Eliminar" con el método "removeElementAt" del bean DefaultListModel :

Haga clic derecho sobre el botón "Eliminar" , luego "Connect" y elija "actionPerformed"
Luego haga clic sobre el bean DefaultListModel.
Seleccione "Connectable Features". Haga clic en "removeElementAt(int)". La conexión aparecerá también discontinua, indicando que está incompleta.

- Complete la especificación de "Agregar" mediante una conexión Parámetro-de-Propiedad para indicar que será el texto del campo de texto el que se agregará a la lista :

Haga clic derecho sobre la conexión evento-a-método que sale del botón "Agregar"

Elija la opción "Connect" y luego la opción "obj".
Luego seleccione el campo de texto y elija "text" del menú que aparece.
La conexión Evento-a-Método pasa a ser una línea continua.

- Complete la especificación de "Eliminar" mediante una conexión Parámetro-de-Propiedad para indicar que será el elemento seleccionado de la lista el que se eliminará de la misma.

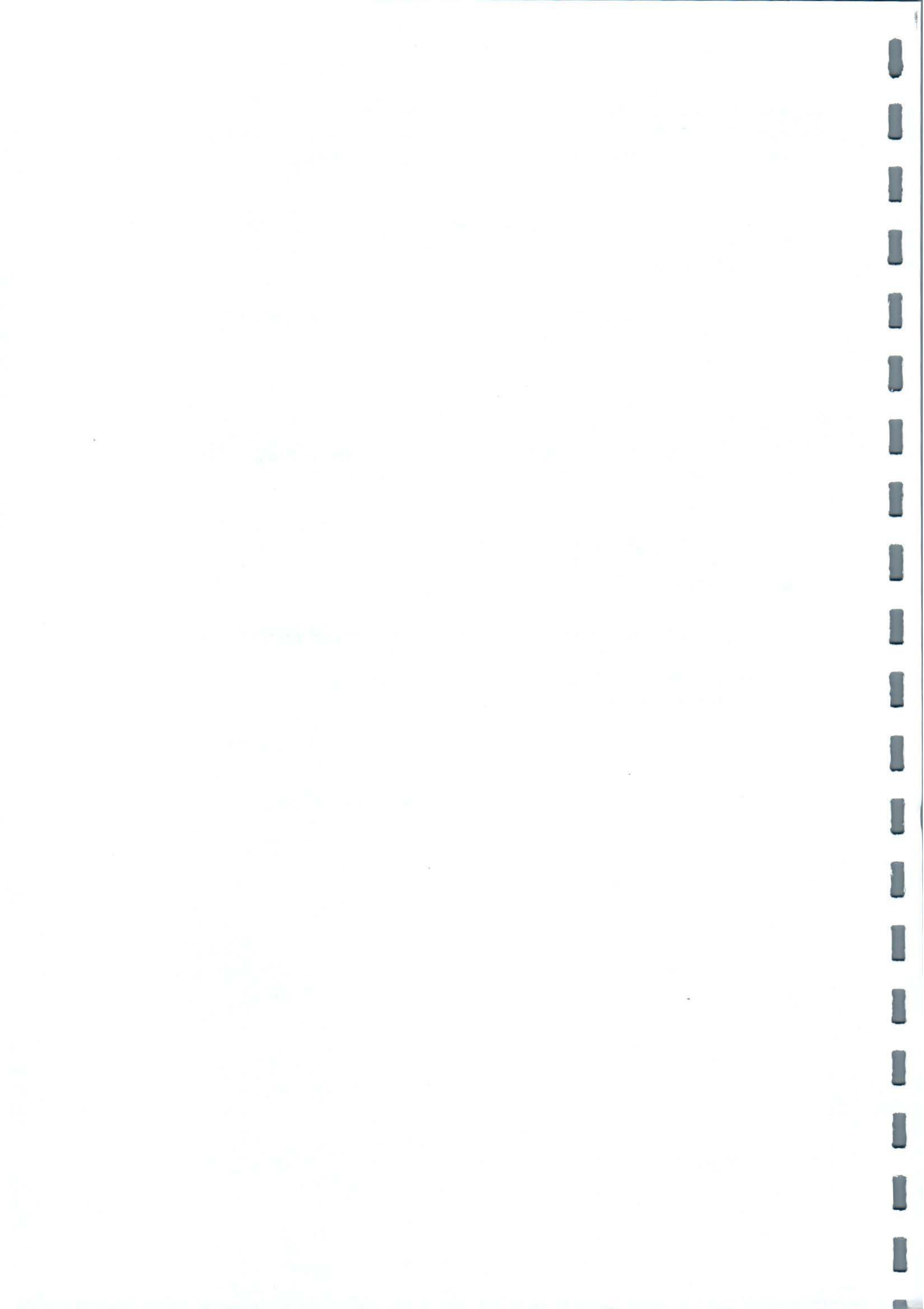
Haga clic derecho sobre la conexión evento-a-método que sale del botón "Eliminar"
Elija la opción "Connect" y luego la opción "index".
Luego seleccione la lista y elija "selectedIndex" del menú que aparece.

- Grafique la conexión que sirva para especificar el envío de datos desde la fuente DefaultListModel a la vista de los datos: El bean JList.

Haga clic derecho sobre el ícono del DeafulListModel.
Elija "Connect" y luego "this".
Haga clic sobre la lista y elija "model"

Aparecerá un conector azul indicando la conexión completa.

Guarde el bean eligiendo la opción "Save bean" del menú "Bean".



Laboratorio 10

Objetivos :

Bases de datos
- Creación de una base de datos
- Creación de una tabla
- Carga de datos con Exportación

JDBC
- Carga de driver JDBC
- Conexión a una Base de datos
- Ejecución de sentencias SQL
- Barrido de resultados extraídos de bases de datos

Tema :

1.- Compruebe que el DB2 esté instalado y en caso de no estarlo, instálelo.

2. Cree la base de datos EMPRESA
Sugerencias :
Arranque el servidor de administración desde la ventana de mandatos
Cree la base de datos desde el Centro de Control

3.- Conéctese a la base de datos EMPRESA desde el Centro de Control.

4.- Compruebe si existen aplicaciones conectadas a esta base de datos.

5.- Cree una fuente de datos ODBC para la conexión con la base de datos EMPRESA

6.- Cree la tabla DEPARTAMENTO con los siguientes atributos (columnas) :

COLUMNA	DESCRIPCION
IdDep	Identificador del Departamento
NombreDep	Nombre del Departamento
Responsable	Identificador de la PERSONA responsable (superior en mando)
DepSuperior	DEPARTAMENTO de quien depende el departamento (el superior en jerarquía organizacional)

<u>Sugerencia</u> :
Desde el centro de mandatos ejecute el comando CREATE TABLE con las columnas correspondientes y con tipos válidos del DB2UDB.

7.- "Cargue" la tabla DEPARTAMENTO con datos. Hágalo importándolos desde el archivo departamento.ASC.

8.- Compruebe que los datos hayan sido cargados, mediante una instrucción SQL.

9.- Conéctese a la base de datos EMPRESA desde un programa en Java y usando un driver JDBC. Hágalo controlando la excepción. Para esto incluya un aviso de "CONECTADO" dentro del bloque controlado.

10.- Muestre el contenido de la tabla departamento, desde una aplicación Java, usando un driver JDBC y una sentencia DML de SQL.

<u>Sugerencias</u> :

Cuando una aplicación se conecta a una base de datos a través de JDBC crea un objeto tipo (que implementa la interfaz) . Usando el objeto de tipo *Connection* con el que estableció la conexión, cree un objeto tipo *Statement* .
Ejecute la sentencia con el método *executeQuery*, que dará como resultado un objeto de tipo *ResultSet*.
Utilice el método *next()* para avanzar a la siguiente fila del resultado.

Para mostrar todas las filas implemente un bucle para, en cada fila imprimir cada columna (use el método print para no cambiar de linea al imprimir).

Métodos útiles para la solución :

getColumnCount() proporciona la cantidad de columnas del resultado.

getColumnDisplaySize obtiene el tamaño (caracteres) de la columna cuya posición es pasada como parámetro.

getString(parámetro) devuelve el valor de la columna *parámetro* en la fila donde se encuentra el cursor. Lo devuelve como un *StringBuffer* de Java.

newInstance puede originar excepciones *InstantiationException* y *IllegalAccessException* .

Solución :

1.- Compruebe que el DB2 esté instalado y en caso de no estarlo, instálelo.

Instale el producto siguiendo la secuencia indicada durante el proceso.

Verifique el grupo de programas instalados por el DB2

Ver diapositivas anexadas

2. Cree la base de datos EMPRESA

Arranque el servidor de administración

Clic al botón Inicio, coloque el puntero en "Programas" y luego en el grupo "DB2 para Windows 95". Seleccione Ventana de mandatos.
Desde la Ventana de mandatos ejecute "DB2ADMIN START"

2.2 Arranque el Centro de Control :
Clic al botón Inicio, coloque el puntero en "Programas" y luego en el grupo "DB2 para Windows 95". Clic en "Herramientas de administración" y luego en "Centro de control".

2.3. Verifique la instancia creada por la instalación
Verifique el lado izquierdo del centro de control. En este panel de objetos expanda el árbol de objetos haciendo clic en los signos "+" hasta llegar a la instancia "DB2" en el árbol de objetos.
(Ver diapositivas anexadas)

2.4 Cree la base de datos EMPRESA

Expanda para ver el contenido de la instancia DB2 y encontrará una carpeta "Bases de datos"
Haga clic derecho sobre "Bases de datos", elija la opción "Crear" y luego la opción "Nuevo"
Ingrese el nombre EMPRESA.

Seleccione la pestaña "Tablas de usuario" y deje el botón de selección exclusiva en "Bajo nivel de mantenimiento" para Gestión del espacio.
Seleccione la pestaña "Tablas de catálogos" y deje el botón de selección exclusiva en "Bajo nivel de mantenimiento" para Gestión del espacio.
Seleccione la pestaña "Tablas temporales" y deje el botón de selección exclusiva en "Bajo nivel de mantenimiento" para Gestión del espacio.
Haga clic en "Terminado" y espere.

3.- Conéctese a la base de datos EMPRESA desde el Centro de Control.

Haga clic derecho en la base de datos EMPRESA en el árbol de objetos y elija "Conectar...". Haga clic en "Bien".

4.- Compruebe si existen aplicaciones conectadas a esta base de datos.

Desde el Centro de Control haga clic derecho en la instancia "DB2" y elija la opción "Listar aplicaciones ..."
Verifique que la aplicación db2cc.exe (precisamente el Centro de Control) está conectada a la base de datos EMPRESA. El id de autorización corresponde a su usuario.

5.- Cree una fuente de datos ODBC para la conexión con la base de datos EMPRESA

Minimice todas las aplicaciones (no salga del Centro de Control ni del Centro de mandatos, sólo minimícelos).
Haga clic en el botón de "Inicio" del escritorio de Windows y elija la opción "Configuración" y luego "Panel de Control".
Ejecute el programa "32bit ODBC" haciendo doble clic en el icono respectivo.

En la pestaña User DSN haga clic en "Add..." y elija "IBM DB2 ODBC DRIVER" de la lista proporcionada haciendo clic. Luego presione "Finalizar". Aparecerá una ventana donde deberá elegir la base de datos que deseamos sea la fuente de datos, en este caso EMPRESA.

6.- Cree la tabla DEPARTAMENTO con los siguientes atributos (columnas) :

COLUMNA	DESCRIPCION
IdDep	Identificador del Departamento
NombreDep	Nombre del Departamento
Responsable	Identificador de la PERSONA responsable (superior en mando)
DepSuperior	DEPARTAMENTO de quien depende el departamento (el superior en jerarquía organizacional)

Desde el centro de mandatos ejecute el siguiente comando de definición (DDL de SQL) :

```
CREATE TABLE DEPARTAMENTO (
       IdDep                    CHAR(3) NOT NULL,
       NombreDep                VARCHAR(40),
       Responsable              CHAR(6),
       DepSuperior              CHAR(3),
       PRIMARY KEY (IdDep))
```

7.- "Cargue" la tabla DEPARTAMENTO con datos. Hágalo importándolos desde el archivo departamento.ASC.

Desde el Centro de Control del DB2 :

Haga clic en "Tablas" y luego clic derecho sobre la tabla DEPARTAMENTO.

Elija la opción "Importar..."

En la pestaña "Archivo" indique el nombre incluyendo la ruta completa del sistema de directorios del ambiente operativo. Por ejemplo c:\datos\departamento.asc

Elija el botón exclusivo "Formato ASCII delimitado (DEL)" para importar tipos de archivos.

Seleccione la opción "INSERT" de la lista de Modalidad de importación.

Escriba "error" en el campo para Archivo de mensajes

Haga clic en "Bien" y espere.

8.- Compruebe que los datos hayan sido cargados, mediante una instrucción SQL.

Desde el Centro de mandatos ejecute la instrucción DML de SQL :

```
SELECT * FROM departamento
```

9.- Conéctese a la base de datos EMPRESA desde un programa en Java y usando un driver JDBC. Hágalo controlando la excepción. Para esto incluya un aviso de "CONECTADO" dentro del bloque controlado.

Sugerencia : Use las sentencias try y catch de Java para manejar excepciones. No olvide cerrar la conexión.

```
import java.sql.*;

public class ConexionJDBC {

 public static void main (String args []) {
        try {
                Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();

                String url = "jdbc:db2:empresa";

                Connection con = DriverManager.getConnection(url,"usrid","psswd");

                System.out.println("conectado a la base de datos");

                con.close();
        }
        catch(Exception e) {
                System.out.println(e);
        }
 }
}
```

10.- Muestre el contenido de la tabla departamento, desde una aplicación Java, usando un driver JDBC y una sentencia DML de SQL.

```
public static void main(java.lang.String[] args) {
        try {
                int n, i = 0;
                StringBuffer dato;
                Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
                Connection con =
                        DriverManager.getConnection("jdbc:db2:sample","usrid","pwd");
                Statement   st = con.createStatement();
                ResultSet   rs = st.executeQuery("SELECT * FROM ORG");
                ResultSetMetaData md = rs.getMetaData();
                int columnas = md.getColumnCount();
                while (rs.next()) {
                        for (n=1; n<=columnas; n++) {
                                dato = new StringBuffer(rs.getString(n));
                                dato.setLength(md.getColumnDisplaySize(n)+1);
                                System.out.print(dato);
                        }
                        System.out.print("\n");
                        i++;
                }
                System.out.println("\n"+i+" filas seleccionadas");

                rs.close();
                st.close();
                con.close();
        }
        catch (ClassNotFoundException   e) {System.out.println(e);}

        catch (SQLException             sqle) {System.out.println(sqle);}

        catch (InstantiationException  ie) {System.out.println(ie.toString());}

        catch (IllegalAccessException iae) {System.out.println(iae.toString());}
}
```

Laboratorio 11

Objetivos :

Beans

Programación Visual
Editor de Composición Visual

Data Access Builder
Utilización de los beans generados para desarrollar aplicaciones

Tema :

Desarrollar una aplicación que muestre una fila a la vez de la tabla DEPARTAMENTO para que pueda realizarse una de las siguientes operaciones sobre la misma :

Agregar una nueva fila
Eliminar una fila
Consultar una fila
Modificar algunos datos de una fila

Las operaciones se deben ejecutar al hacer clic sobre uno de los cuatro botones ofrecidos para cada una de las operaciones mencionadas.

Utilice las facilidades del Data Access Builder para generar beans y usar éstos desde el editor de composición visual de Visual Age for Java.

Los botones que pueden utilizarse pueden ser de la categoría AWT o de la categoría swing.

Solución :

Cree un proyecto (project) en el workbench de VAJava. Por ejemplo en la diapositiva anexada puede apreciar que se creó el proyecto "DABeans".

Crea un paquete (package) en ese proyecto. Por ejemplo en la diapositiva anexada puede apreciar que se creó el paquete "Beans".

"Arranque" el Data Access Builder (DAB) en el paquete.

Para "arrancar" el Data Access Builder se debe :

Desde el Workbench, seleccionar el paquete que creó y elegir de su menú de contexto.
Para acceder al menú de contexto de un objeto, como un paquete en este caso, puede darse clic derecho sobre el mismo (ver la diapositiva anexada) o también puede usarse el menú "Selected" del Workbench.

Elegir la opción Tools
Elegir la opción Data Access
Elegir la opción Create Data Access Beans...

Al elegir esta opción se arranca el Data Access Builder, que en un primer momento sólo presentará al ícono correspondiente al paquete.

Estando en la ventana del Data Access Builder (indicada con el número 1 en la diapositiva) ...

Para crear un "mapping" con un esquema de BD elija la opción "Map Schema" del menú de contexto del paquete (menú Selected de la barra de menú o clic derecho para el menú "pop-up").

Haga clic en el botón "Next" (siguiente) para ir a la ventana de selección de base de datos y método de mapping (indicada con el número 2 en la diapositiva).

En esta ventana el DAB permite especificar la base de datos fuente, que en este caso es DB2 y la base de datos particular. Elija la base de datos "EMPRESA".

En esta misma ventana el DAB permite elegir entre dos métodos para el mapping :

Seleccione el correspondiente a Seleccionando las tablas/vistas directamente de la base de datos elegida : la opción "By selecting database tables and views".

Haga clic en el botón "Next" y accederá a la ventana de selección de tablas (indicada con el número 3 en la diapositiva anexa) donde deberá especificar la tabla DEPARTAMENTO.

En la ventana del Data Access Builder se apreciará el bean DEPARTAMENTO y sus atributos.

Puede apreciarse un ícono diferente para el atributo llave primaria del ejemplo mostrado.

Verifique el driver JDBC que se utilizará y la ubicación (URL) de la base de datos. Haga clic en la opción de propiedades del menú de contexto del bean Departamento.

Para generar los beans y clases que podrán ser usados para construir aplicaciones con los objetos de base de datos especificados, seleccione el menú "File" y la opción "Save and Generate".

Los fuentes de Java son automáticamente importados y compilados en el paquete.

Se generaran 18 clases que encapsulan el acceso a la base de datos para aplicaciones que lo requieran (ver diapositiva anexada).

Cree un paquete diferente para contener la aplicación. Por ejemplo el paquete "Aplicaciones"

Cree una clase indicando que la construirá visualmente. Colóquele un nombre adecuado, por ejemplo "Departamento".

Automáticamente se presentará el editor de composición visual (si no ubíquelo desde el visor de la clase Departamento).

En el Editor de Composición Visual integre los beans generados a la paleta de beans :

Para esto debe accederse al menú Bean y elegir la opción "Modify Palette".

En esta ventana puede incorporarse beans a grupos existentes :

Primero debe crear el grupo llamado EMPRESA, tal como puede apreciarse en la diapositiva

Incorpore el bean DepartamentoForm en el grupo creado.

Utilice la facilidad de búsqueda (clic en botón "Browse") y seleccione la clase correspondiente desde la ventana "Choose a valid class".

Finalmente haga clic al botón "Add to Category".

Incorpore de la misma manera los beans :

Departamento
DepartamentoDatastore

Los botones que pueden utilizarse pueden arrastrarse de la categoría AWT.
Seleccione la categoría AWT y arrastre los botones.

Modifique cada uno de los botones, cambiando la propiedad de texto, a los apropiados : "Agregar", "Eliminar", "Consultar" y "Actualizar".

Del grupo EMPRESA arrastre el componente DepartamentoForm.

Del grupo EMPRESA arrastre el componente Departamento de manera que la aplicación interactuará con el objeto de base de datos : tabla Departamento (cuyos objetos precisamente representan filas de la tabla).

Las acciones que se desea realice la aplicación son las correspondientes a los botones.

Conecte cada uno de los botones con conexiones tipo "Evento-a-Método" :

Botón	Evento	Método
Agregar	actionPerformed	add
Eliminar	actionPerformed	delete
Consultar	actionPerformed	retrieve
Modificar	actionPerformed	update

Arrastre los beans no visuales :

DepartamentoDatastore cuyo objeto representa la conexión a la base de datos y se requerirá precisamente para conectarse, desconectarse y comprometer o deshacer las transacciones sobre la BD.

MessageBox, bean para ser usado en el despliegue de mensajes de error.

Este bean se encuentra en la categoría (grupo en paleta) "Enterprise Access".

Entonces para establecer la conexión a la base de datos apenas se cargue el applet, establezca una conexión evento-a-método entre el evento *init()* del applet y el método *connect()* de bean DepartamentoDatastore.

Para controlar una posible falla o imposibilidad de conexión con la base de datos, conecte esta última conexión con el bean MessageBox. Para esto establezca una conexión evento-a-método entre el evento *exceptionOcurred* y el método *showException* del bean MessageBox.

Para indicar el texto del mensaje, "abra" esta conexión (doble clic o "Open" en el menú de contexto) y establezca el indicador "Pass event data" (ver diapositiva).

De igual forma que para la conexión, deberá indicarse la funcionalidad para la desconexión de la base de datos, que debería ocurrir al salir del applet.

Para esto sólo establezca una conexión evento-a-método desde el evento *destroy* del applet y el método *disconnect* del DepartamentoDatastore.

Para mostrar cualquier falla en las operaciones correspondientes a los cuatro botones de acción, establezca conexiones entre las conexiones que van de cada botón al bean Departamento con el bean MessageBox. Para esto establezca una conexión evento-a-método entre el evento *exceptionOcurred* y el método *showException* del bean MessageBox para cada conexión.

Para indicar el texto del mensaje, debe "abra" cada conexión (doble clic o "Open" en el menú de contexto) y establezca el indicador "Pass event data" (ver diapositiva).